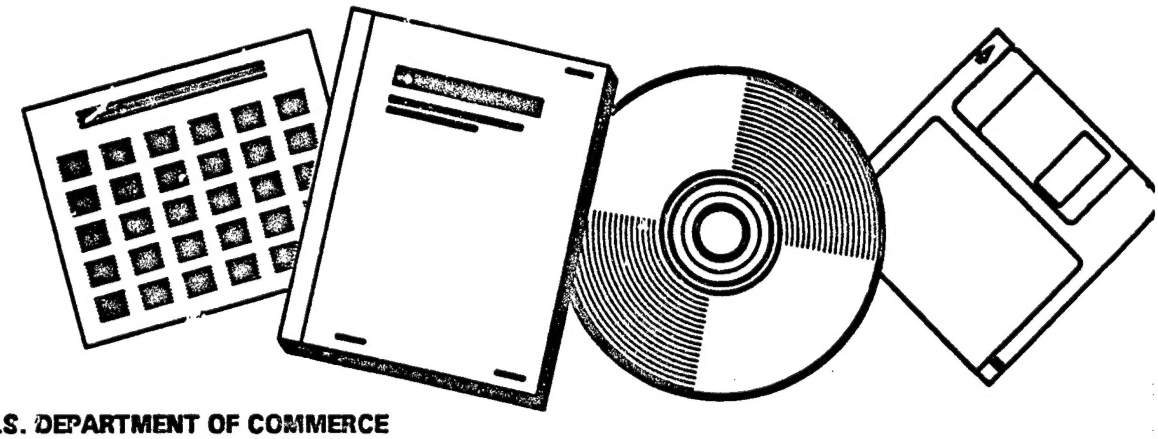
PB96-148408

COMPLEAT GUIDE TO MRS

STANFORD UNIV., CA

JUN 85



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

BIBLIOGRAPHIC INFORMATION

PB96-148408

Report Nos: STAN-CS-85-1080, KSL-85-12

Title: Compleat Guide to MRS.

Date: Jun 85

Authors: S. Russell.

Performing Organization: Stanford Univ., CA. Dept. of Computer Science.

Sponsoring Organization: *Office of Naval Research, Arlington, VA.

Contract Nos: ONR-N00014-81-K-0004

NTIS Field/Group Codes: 62 (Computers, Control & Information Theory), 62B (Computer Software)

Price: PC A07/MF A02

Availability: Available from the National Technical Information Service, Springfield, VA. 22161

Number of Pages: 129p

Keywords: *Logic programming, Artificial intelligence, LISP programming language, Databases, Reasoning, Knowledge bases(Artificial intelligence), Inference, *MRS(Meta-level Representation System), *Meta-level Representation System.

Abstract: Metal-level Representation System (MRS) is a logic programming system with extensive meta-level facilities. As such it can be used to implement virtually all kinds of artificial intelligence (AI) applications in a wide variety of architectures. This guide is intended to be a comprehensive text and reference for MRS. It also attempts to explain the foundations of the logic programming approach from the ground up, and it is hoped that it will thus provide access, even for the uninitiated, to all the benefits of AI methods. The only prerequisites for understanding MRS are a passing acquaintance with LISP and an open mind. The first part of the book deals with the principles and basic commands of MRS, and is sufficient to allow the reader to begin creating fairly complex systems. The second part covers the advanced features of MRS that enable the user to tailor the system to her own needs and increase performance functionality as desired.

June 1985

Report No. STAN-CS-85-1080
Also numbered KSL-85-12

PB96-148408

The Compleat Guide to MRS

by

Stuart Russell, Esq.

Department of Computer Science

Stanford University
Stanford, CA 94305

Stanford Knowledge Systems Laboratory
Report No. KSL-85-12

June 1985

The Compleat Guide to MRS

by

Stuart Russell Esq.

COMPUTER SCIENCE DEPARTMENT

Stanford University
Stanford, California 94305

This work was funded from ONR contract N00014-81-K-0004

Preface

MRS is a logic programming system with extensive meta-level facilities. As such it can be used to implement virtually all kinds of artificial intelligence applications in a wide variety of architectures. This guide is intended to be a comprehensive text and reference for MRS. It also attempts to explain the foundations of the logic programming approach from the ground up, and it is hoped that it will thus provide access, even for the uninitiated, to all the benefits of AI methods. The only prerequisites for understanding MRS are a passing acquaintance with LISP and an open mind.

The first part of the book deals with the principles and basic commands of MRS, and is sufficient to allow the reader to begin creating fairly complex systems. The second part covers the advanced features of MRS that enable the user to tailor the system to her own needs and increase performance and functionality as desired. The best way to read the book is to try out everything on the terminal as soon as it is introduced, or even to be developing an application concurrently with learning the material. There are several exercises provided which are not the least bit optional. They are however non-trivial, so don't be alarmed if some of them seem daunting. In the author's opinion, the style of programming induced by MRS is far more natural and uncomplicated than traditional methods, and it is only the corrupting influence of previous education that makes logic programming seem a little strange at first.

The author would like to thank Prof. Michael Genesereth and the other authors of MRS; Russ Greiner, Matt Ginsberg, Leonor Abraido-Fandiño, Ben Grosof and René Bach for numerous useful suggestions and diligent reading; and Eric Berglund for help with the figures.

CONTENTS

Part I :

Understanding MRS

Chapter 1
Introduction

What is MRS? Good question. MRS stands for Meta-level Representation System. If your response to this is a knowing nod of understanding you can probably skip the first few chapters. In a sense, MRS is a computer language, in that one enters text in a designated syntax and it gets processed and produces answers (or not). But because MRS is also able to reason with the information you give it, the "program" you enter can be seen more as representing facts than specifying a process. The importance and utility of this difference will become clear.

§1.1 Problems problems problems

Like all computer languages, MRS is a tool for solving problems. By the time you have mastered it, you will know how to use it to solve some Very Difficult Problems Indeed. But unlike most computer languages, MRS doesn't require that you know exactly how to solve a problem before it can help you. This distinction is not iron-clad — if the claims of AI have any merit then Turing-equivalence assures us that Albert Einstein could be implemented in BASIC too — but the normal style of problem-solving in MRS is radically different from that in FORTRAN or even LISP. Yes, even today people say things like "The great thing about computers is that they make sure you know *exactly* how to solve a problem." You may or may not agree that this is a great thing, but with the advent of MRS (and, I suppose, some other systems too) it's no longer true.

The (somewhat idealised) traditional process of computer use goes something like this:

1. Identify the problem.
2. Assemble what you know about it.
3. Decide on data structures to correspond to things in the problem.
4. Figure how to process those structures to produce the desired answer.
5. Encode the process step by step in your favourite language.
6. Get the computer to execute the process.

whereas the happy MRS user can just do the following:

1. Identify the problem.
2. Assemble what you know about it.

3′. Encode what you know about it in MRS.

4′. Get MRS to figure out the solution.

That, at least, is the plan. The key problem faced by the experienced programmer coming to MRS is that she can't stop herself from doing steps 1–4 from the old tradition, then trying 5 in MRS, whereupon she concludes that it's no better than Pascal, has a lot of silly parentheses and doesn't even have loops.

The new regimen places more emphasis on stage 2, assembling what you know about the problem, which is often overlooked in the old ways with disastrous results in 6. It should already be clear that less effort is really required in the basic procedure for solving problems in MRS; but as any Real World Programmer will tell you, that's not the whole story. Debugging and maintenance are apparently non-trivial affairs. Debugging is changed as follows: clearly if all you have is facts, then you can only get the wrong answers if some of the facts are false. Not only is it unlikely that you would type a false fact in the first place (apart from typographical inaccuracies), but it is also considerably easier to notice that a fact is false than it is to decide exactly how it is that the twelve-page procedure you have just coded doesn't correspond to the solution process you had in mind, even if the solution process was correct in the first place. Maintenance is also considerably eased: if

some facts about the problem change, the enlightened MRS user changes the corresponding facts in his system; the traditional programmer cusses loudly, tries to find all those places in the program where the solution process relied on the truth of the original facts, and prays that the changes won't have more than the usual number of side-effects.

Given that MRS isn't yet Albert Einstein, this has been an idealized version of the truth, but it illustrates the basic approach that should be followed in producing applications using MRS.

§1.2 Logic doesn't rhyme with Magic

You may suspect that the stage 3' above, "Encode these facts in MRS", is not quite as straightforward as all that. But facts are being encoded all the time; in fact you're reading fact-encodings right now (or not, depending on your philosophical inclinations). To see that a fact is different from a sentence, recall that the same fact could be expressed with different sentences, e.g.

John is taller than Ken
Ken is shorter than John

or even

Ken est plus petit que John.

Some of the linguistic philosophers in the audience may disagree with it, but this simplistic approach is all that's necessary for understanding MRS.

For centuries, philosophers have tried to devise formal languages for encoding facts about the world so that strict rules could be applied for deriving new facts from old. In this way, valid arguments could be distinguished from invalid. The schemes of propositional and predicate calculus, originating in the work of Frege, are the currently accepted standard. The important thing about these schemes is not the particular syntax employed or the applicable rules of inference, but their way of viewing the world.

According to predicate calculus, the universe, as it's often called, consists of a fixed set of objects (not necessarily finite). This universe is not usually the same as the whole Universe As We Know It; in fact it's often a very small subset, for instance {John,Ken}. A key notion is that we have independent access to the objects — we know what we mean by 'John'. The objects need not be 'real' things: unicorns, numbers, sets and Pepsis can inhabit universes too.

In the formal language, we will have *constant symbols* to refer to these objects — names, if you like. For example, I might use the constant symbols John and Ken or j12 and k535 to refer to the objects John, Ken in the universe. Notice that even in English we still have to use such symbols; strictly I should insert the real John and Ken into the text, but even modern computerised typography doesn't have such fonts. The point is that the idea of using symbols to refer to actual objects is extremely natural.

In English, we often want to refer to objects that don't have names of their own, such as John's left foot. Similar constructions are used in MRS, and are called *terms*. Terms are written with a *function symbol* followed by its arguments, which are also terms; a constant symbol is a kind of term. The whole lot is enclosed in parentheses. So we might write (`LeftFootOf John`) or (`LeftFootOf (FatherOf Ken)`) but probably not (`FootOf Ken`) unless we were in a world of molluscs: a term refers to a unique object. It is essential to remember that terms are just fancy kinds of names; they are *not* computable expressions.

Along with objects, there are relations. These enable you to say things about the objects, and in fact that is basically all that can be said. In my universe {John,Ken} I might want to have the relation of being taller than. Now whatever I might think about the meaning of such a relation,

according to *relational semantics* a relation is defined by the set of all tuples of objects which satisfy the relation (its *extension*). Thus, suppose my *relation symbol* for this relation is `IsTallerThan`. Then in the {John,Ken} universe,

`IsTallerThan` ≡ {⟨John,Ken⟩}.

Now if John is also older than Ken,

`IsOlderThan` ≡ {⟨John,Ken⟩}

so `IsTallerThan` and `IsOlderThan` are the same relation. Good grief! Actually, this isn't just plain daft, it's a salutary lesson in the art of expressing facts: from the point of view of a system such as MRS containing such facts, the two relations are identical since one could exchange the two relation symbols throughout and not affect the result of any computation. In fact you could replace `IsTallerThan` by `G00062` and still not change anything. The system *really does not know that John is taller than Ken.* It just knows that `IsTallerThan` is a binary relation which holds between John and Ken. For many purposes, however, it is useful to pretend that the system 'knows' these facts, and no doubt this grave error is committed liberally throughout this book.

Notice that relations come in many different kinds: unary relations like

`PlaysBaseball` ≡ {⟨*John*⟩, ⟨*Ken*⟩},

binary relations like `IsTallerThan`, and in fact relations with any number of arguments.

All we need now to start encoding facts in our formal language is a syntax for combining relation symbols with constants to express the fact that the relation holds for the objects referred to by the constant symbols. The syntax used by MRS will be described in the next chapter, along with more apparatus of predicate calculus that enables more complex facts about the universe to be expressed. Chapter 4 will introduce the ideas involved in inferring new facts from old. These ideas are the key to stage 4′ of the MRS programming process, "Get MRS to figure out the solution".

Chapter 2
Representing knowledge in MRS

The preceding chapter introduced the idea of a formal language for expressing facts, but didn't actually express any. The reason for this is that to actually express a fact one needs to choose a syntax; it is important to remember that the syntax used is a somewhat arbitrary choice that must be clearly distinguished from the ideas of representation discussed above. The choice of syntax depends on such things as readability and ease of manipulation by computer programs. Because MRS is a LISP-based system, the syntax is chosen to mesh well with LISP. The following sections discuss the actual syntax of MRS.

§2.1 Symbols

Any LISP atom can be used for constant, function or relation symbols, with a few exceptions:
- atoms beginning with & and $ have special uses which will be explained later;
- the atoms `AND OR NOT IF` should not be used as relations.

§2.2 Ground literals

Generally speaking, the expression of a fact in a formal language is called a *proposition*. The facts that we didn't express in Chapter 1 are called *ground literals* — they express that a given relation holds between the given objects. Given the above hint about LISP, you can probably guess that "John is taller than Ken" is expressed by

```
(IsTallerThan John Ken).
```

Terms with function symbols such as `(FatherOf Ken)` can appear in the same places as constants, so since John is Ken's father,

```
(IsTallerThan (FatherOf Ken) Ken)
```

expresses the same fact. Since terms can be arbitrarily nested, we could say

```
(PlaysBaseball (FatherOf (MotherOf (FatherOf Miranda)))).
```

§2.3 Equality

A relation with a special meaning in MRS is the equality relation =.

```
(= <term1> <term2>)
```

is true if and only if the two terms refer to the same object in the universe. Thus `(= John John)` is automatically true; `(= (FatherOf Ken) John)` is true as long as John is Ken's father. If we tell MRS that

```
(= MorningStar EveningStar)
```

then there will be two constant symbols that refer to the same object, but unless otherwise specified, MRS will assume that different constants refer to different objects, so they are not necessarily =. Moreover, MRS is equally unable to decide that different constants *necessarily* refer to different objects, so they are not necessarily not = either.

§2.4 More complex propositions

Central to the utility of predicate calculus is the idea that complex facts can be expressed as combinations of simpler facts, the modes of combination being such that the truth of a complex fact can

be determined from the truth of its constituents. Thus, for example, the fact "John is taller and older than Ken" is seen as a *conjunction* of two simpler facts, and is true if and only if "John is taller than Ken" and "John is older than Ken" are both true. Notice that the same property does not hold for sentences such as "John is taller than Ken because John is older than Ken": even if both parts are known to be true, say, we still don't know if the whole sentence is true or false because 'because' refers to a context of causal relations outside the sentence itself.

Complex facts are expressed in predicate calculus using *logical connectives*. MRS recognizes the logical connectives AND, OR, NOT and IF:

(AND <proposition$_1$ > ... <proposition$_n$ >)
: is true when all the conjoined propositions are true. A conjunction of no propositions is true.

(OR <proposition$_1$ > ... <proposition$_n$ >)
: is true when any of the disjoined propositions are true. A disjunction of no propositions is false.

(NOT <proposition>)
: means that the proposition is false.

(IF <antecedent proposition> <consequent proposition>)
: is true if the consequent is true when the antecedent is true, or if the antecedent is false; i.e. the consequent follows from the antecedent.

§2.5 Variables

Many facts that we might want to express are general in the sense that they do not refer to any particular object or objects in the universe, but have truth conditions depending on the truth of all the individual propositions generated by applying the statement to each particular object in the universe. For example, in the $\{John, Ken\}$ universe, the statement "All Americans play baseball" can be expressed by

```
(AND (IF (American John) (PlaysBaseball John))
     (IF (American Ken) (PlaysBaseball Ken)))
```

and it's truth decided by using the rules for AND and IF. But for larger universes this becomes a tedious way to express what seem to be quite simple statements, and for statements like "The sum of any two integers is an integer" the task of enumerating all propositions of the form

```
(IF (AND (Integer 1) (Integer 2))
    (Integer (+ 1 2)))
```

is really quite lengthy.

Several other kinds of statement are commonly used which have similar properties:

"There are some Americans who don't play baseball."

"At least half of all Americans forget to vote."

"There are two hundred million Americans."

MRS provides a simple mechanism for expressing only *universal* propositions, which state that some fact is true of every object in the universe. Of those mentioned, these are by far the most commonly needed type. All you do is use a special symbol type, called a *variable*, in the place where the constant symbol would go if the general statement were applied to a particular object. A variable is just a symbol beginning with a dollar sign $, e.g. `$X $$$ $variable`. So the statement "All Americans play baseball" is expressed as

```
(IF (American $X) (PlaysBaseball $X))
```

"Everything is going wrong today" as (literally speaking)

```
(GoingWrongToday $Y)
```

"The sum of any two integers is an integer" as

```
(IF (AND (Integer $X) (Integer $Y))
    (Integer (+ $X $Y)))
```

There are two highly important things to note about variables.

Firstly, the particular variable name you use is irrelevant (although it may have mnemonic value). Thus, to MRS, (GoingWrongToday $X) and (GoingWrongToday $everything) are the same fact.

Secondly, where variables appear in more than one place in a proposition, it matters whether they are the same or different. Thus

```
(IF (AND (Integer $X) (Integer $X))
    (Integer (+ $X $X)))
```

means something very different from the proposition above that also uses $Y. It says that when you add an integer *to itself* you get another integer. Thus the key observation is that whenever a variable occurs more than once in a proposition, each occurrence will refer to the same object.

Unlike many systems of predicate calculus, computerised or not, MRS allows variables in place of functions and relations. At first this may seem a little unnecessary — we don't often want to say things like ($X Fred), meaning that every unary relation is true of Fred. But in restricted universes where we are dealing with a known set of relations, this might be useful. Also we can use this facility to describe classes of relations:

```
(IF (Reflexive $R) ($R $X $X))
```

says that every object is related to itself by any reflexive relation.

§2.6 Existential propositions

In the previous section we said that MRS had no simple way to handle anything other than universal propositions. This isn't quite true. Consider the case of *existential* propositions, i.e. propositions which state that some fact is true of at least one object in the universe. An example is given above: "There are some Americans who don't play baseball". This can be interpreted as a statement about some unknown individual, whose sole properties are that she is American and doesn't play baseball; we can invent a name for this individual such as LUH9781, and express the existential proposition by

```
(AND (American LUH9781) (NOT (PlaysBaseball LUH9781))).
```

It is essential that the name used for the individual be *new* in the sense that no other facts in the database thus far can have mentioned it. Given this condition, you should be able to convince yourself that nothing essential is lost in the translation. The general name for this process is *skolemization*, but the general process for arbitrarily nested universal and existential propositions is too complex to be given here. Any good textbook on predicate calculus should contain an adequate description.

§2.7 Exercises in representation

In these exercises, the given symbols have the obvious meanings; where you are asked to write your own propositions, use an equally perspicuous vocabulary.

Express the following propositions in reasonable English:

1. (Horse Dobbin)
2. (Member Dobbin Horses)
3. (NOT (Horse Dobbin))
4. (OR (Horse Dobbin) (Donkey Dobbin))
5. (IF (Horse Dobbin) (Mammal Dobbin))
6. (IF (Horse $x) (Mammal $x))
7. (IF (OR (Horse $x) (Cow $x)) (FourLegged $x))
8. (AND (Mammal $x) (FourLegged $x))
9. (IF (NOT (Horse Dobbin)) (Dutchman Ermintrude))
10. (IF (NOT (Cow $x)) (Brown Dobbin))
11. (IF (AND (Horse $x) (NOT (Mammal $x))) (Cow $x))
12. (IF (OR (= $x Dobbin) (= $x Tonto)) (Horse $x))
13. (IF (AND (= $x Dobbin) (= $x Tonto)) (Horse $x))
14. (IF (AND (Mammal $x) (NOT (= $x Dobbin))) (NOT (Horse $x))
15. (IF (AND (Horse $x) (NOT (= $x $y))) (NOT (Horse $y))
16. (NOT (Member $x $x))
17. (IF (AND (Horse $x) (Brown $x)) (Brown (TailOf $x)))

Express the following English in reasonable propositions:

18. All dogs bark at their neighbours' dogs.
19. No real numbers are integers.
20. Horses who hate dogs like ice cream.
21. Giraffes have longer necks than Dobbin.
22. An-An is the only male panda in London.
23. Zero is an integer.
24. The fractional part of an integer is zero.
25. The product of two real numbers is a real number.
26. The product of a positive integer and its inverse is unity.
27. Zero is an additive identity. (Don't say (AdditiveIdentity 0)!)
28. The product of two real numbers is never an imaginary number.
30. All numbers are either real or imaginary or both.
31. All Englishmen, Scotsmen and Welshmen are British.

Chapter 3

Storing and retrieving facts — the database

Once all these facts are represented, we must tell them to MRS, which must store them away somewhere to be used in solving problems. Obviously, we should like to be able to inspect and change the facts at will.

MRS stores facts in its *database*, also occasionally known as the knowledge base or rule base. Think of the database as an infinitely extensible repository of knowledge, so organised that the time taken to retrieve a fact from it is essentially independent of the number of facts it contains. But before we talk about the MRS commands for storing and retrieving facts, we must understand the concept of a *query*.

§3.1 Queries

In MRS, as in all logic programming, problem solutions are given by facts. For instance, if I want to sum the numbers in a list L then the desired solution will be a fact of the form "The sum of the numbers in L is N". Database retrieval is a simple form of problem-solving, one in which the answer to the problem is a fact already in the database.

Suppose we have a very simple problem: find the age of John. Our desired solution fact will look like (Age John x) for some x. What we need to do is ask the system to prove a fact of this form, given what facts we have already put into its data base. In MRS, as in most logic programming systems, we can pose this as a query (Age John $x). Note that this isn't the same as the fact in the data base, which would mean that everything in the universe was the age of John. It is asking the system to find any $x such that $x is the age of John. The query is in fact treated as an existential proposition to be verified. (Note that, as in the case of representing existential propositions in the database using anonymous constants, we can have universal queries too. For example, if we wanted to show that everyone was 22, we could use the query (Age G00089 22); if the system can prove that an object about which it knows nothing is 22, it can prove the same fact about anything.)

Returning to our simple problem, let's take the case where John's age is already known: (Age John 22) is already in the data base (I said this was a simple problem). The system succeeds in solving the problem by noting that (Age John 22) matches the query, provided we let $x be 22. Such an association of a variable with a constant symbol is called a *binding*. In MRS a binding is represented by a CONSed pair such as ($x . 22). In general, a query could have more than one variable, e.g. (Parents John $father $mother), so a solution is represented by a *binding list* such as (($father . Ian) ($mother . Iris)). In general we say that a query is satisfied by a binding list if the process of substituting the variable bindings from the list into the query produces a fact which is true.

Suppose, however, that we know that everyone in our universe is 22, i.e. (Age $y 22) is in the data base, but John's age is not specifically mentioned anywhere. Obviously, we would like the system to produce the solution. If we remember that (Age $y 22), as a data base fact, is shorthand for (Age Alf 22) (Age Bert 22) . . . (Age John 22) . . . (Age Zack 22), then the answer is obvious: allow bindings for variables in the data base fact as well as in the query. Thus in this case a binding list (($x . 22) ($y . John)) would, if substituted into either of the two propositions (Age $y 22) and (Age John $x), produces the same fact (Age John 22). The substitution into the data base fact produces a fact which is still true; the fact thus produced matches with the query fact as before, so the solution is guaranteed to be valid.

The process of finding a binding list which, when substituted into two propositions, makes them the same, is called *unification*. The binding list is called a *unifier*. Thus one method of trying to solve a problem posed as a query proposition Q is to find a fact P in the data base such that that P

and Q unify, with unifier σ. If we denote the result of substituting a binding list θ into a proposition R as $R\theta$, then we require that $P\sigma = Q\sigma$.

§3.2 Exercises on unification

For each of the following pairs of propositions you are to find the binding list that unifies them (if any). Assume that variables with the same name in different propositions are distinct, so you will have to rename the variables in the second proposition if there is any conflict. Some of the examples contain dot notation — this is the same as the LISP notation for CONS, so for example the propositions

```
(p $w $x (f $y))
```

and

```
(p a . $z)
```

unify with unifier `(($w . a) ($z . ($x (f $y))))`.

1. `(p $a)` and `($r x)`.
2. `(p $a)` and `($a q)`.
3. `(p $a c)` and `(p $y)`.
4. `(q (f $c))` and `(q $d)`.
5. `(r (g $c))` and `(r $c)`.
6. `(r $x (h $x))` and `(r $b $b)`.
7. `(p $a (g $a) (h $a))` and `(p (g $b) (g . $c) ($d . $c))`.
8. `(q $a)` and `($r . $s)`.
9. `(r $b . $b)` and `(r $c a)`.

§3.3 Actually doing things with the MRS database

After such a lengthy introduction, even the most diligent reader is probably itching to release the awesome power of MRS. The monitor-level command you need to invoke MRS is installation-dependent, so we'll assume here that MRS is ready to go.

Commands typed to MRS are actually typed to the LISP interpreter. The normal LISP read-eval-print loop is still in operation All MRS commands are performed by functions coded in LISP, so the ways of entering commands are the same as those for invoking LISP functions: they can be entered from the terminal or read in from a file using the LOAD function. In addition the MRSLOAD command can read in MRS propositions directly from a file (see Ch. 10). As with LISP, text beginning with the comment character ';' is ignored up to the end of the line, so MRS command files can be commented just like programs. Ordinary LISP functions can be invoked at all times, and MRS functions can be invoked from user code.

It is intended that the reader read this chapter sitting at a terminal and type the entries to the right of the > symbol. Studies have shown that those who avoid this become stunted MRS users. Moreover, the facts you enter here will also be used in chapter 4, so it's probably best to work through the two chapters in the same session.

§3.4 Getting facts into and out of the data base

The straightforward and unromantically-named STASH command does the straightforward and unromantic job of adding a new fact to the data base. Don't forget to quote the fact you are stashing.

```
>(STASH '(Parent Alice Bert))
```

```
P108
>(STASH '(Parent Alf Bert))
P109
```

You may find MRS's responses a little puzzling, not to say insulting. What it's actually telling you is that your propositions have been stored on the property lists of the atoms P108 and P109 (the actual numbers may vary). These atoms are called proposition symbols.

```
>(STASH '(Female Alice))
P110
>(STASH '(Male Alf))
P111
>(STASH '(Male Bert))
P112
```

An alternative to STASH is ASSERT, which initially does exactly the same thing. However, often the user will want further inferences to be made automatically from the facts she enters, and ASSERT is used for this, whilst STASH is normally reserved for simple storage of facts.

To retrieve facts from the data base, use the LOOKUP command:

```
>(LOOKUP '(Male Bert))
((T . T))
>(LOOKUP '(Female Alf))
NIL
```

LOOKUP returns the binding list that satisfies the query. If the query contains no variables, it just returns a nominal list ((T . T)) to indicate that the fact was in the data base — this distinguishes the situation from that pertaining when the fact can't be found and LOOKUP returns NIL. To find out the name of Alf's child, type

```
>(LOOKUP '(Parent Alf $x))
(($X . BERT) (T . T))
```

As you can see, MRS has no respect for your careful use of upper and lower case, but don't abandon it because it helps to make your source files much more readable. You may find that variables appear slightly differently in the returned binding lists — this is a LISP effect so don't worry about it.

Clearly, some queries with variables can be satisfied in more than one way, such as (Parent $p Bert). LOOKUP returns the first solution it finds, and the data base is searched in the reverse order from that in which the facts were stashed. This search order is an important, if arbitrary, part of MRS. It can be used to give a kind of 'priority' ranking to facts and rules, and is important in the understanding of how MRS implements various defaults.

To get all the answers to a query, use LOOKUPS which returns a list of binding lists:

```
>(LOOKUPS '(Parent $p Bert))
((($P . ALF) (T . T)) (($P . ALICE) (T . T)))
>(LOOKUPS '(Male $x))
((($X . ALF) (T . T)) (($X . BERT) (T . T)))
>(LOOKUPS '(Parent $p $c))
((($P . ALF) ($C . BERT) (T . T)) (($P . ALICE) ($C . BERT) (T . T)))
```

If by some dreadful mischance you happen to stash a fact that isn't quite true, you can remove it using UNSTASH:

```
>(UNSTASH '(Female Alice))
(FEMALE ALICE)
>(STASH '(Goddess Alice))
P113
```

An extremely useful command is FACTS, which prints all the facts containing its argument (which will usually be an atom, but can be a term):

```
>(FACTS 'Alice)
P108: (PARENT ALICE BERT)
P113: (GODDESS ALICE)
```

FACTS can also take a second, numeric argument indicating the maximum level at which the first argument may appear in a fact for it to be printed (rather like PRINTLEVEL in LISP). See chapter 12 for ways to specify the output format for facts. You can use FACTS to avoid the tedious chore of typing out a whole fact, character by character, simply in order to unstash it. Calling FACTS with an appropriate argument will tell you the proposition symbol for the unwanted fact; then the system function Pattern, which takes a proposition symbol as argument and returns the associated fact, can be used thus:

```
(UNSTASH (Pattern 'P113)).
```

UNASSERT can also be used to remove facts. Like ASSERT, it is normally used when further inference is desired, presumably resulting in the removal of dependent facts.

Chapter 4
Reasoning with Knowledge

§4.1 Rules of Inference

Having entered all these facts, what more can be done with them? How does MRS figure out solutions to problems? The direct answering by LOOKUP of queries on the database can be seen as a simple case of problem-solving, and, as we said in the previous chapter, problem solutions are given by facts. Thus the process of computation in MRS is one of producing new facts from the original facts encoded about the problem. Obviously, we can't just produce any old facts: they should probably follow in some way from the facts already known. The rules which determine what facts can be added from given facts are called *rules of inference.*

A rule of inference is usually written like this:

$$\frac{\text{< description of initially known fact(s) >}}{\text{< facts that can be inferred >}}$$

For example, if we know that (AND A B) is true, we can infer that A is true and B is true:

$$\frac{(AND\ A\ B)}{A} \qquad \frac{(AND\ A\ B)}{B}$$

If you've gone to a lot of effort to make sure the initial facts encoded about the problem are true, then usually you'd like all the facts inferred, in particular the solution fact, to be true too. An important class of inference rules consist of those which guarantee the truth of the inferred facts provided the initial facts are true. A system using just rules of this type is said to be *sound* and is called a deductive system. A system with a set of inference rules which is sufficient to produce *all possible* deductions from a given set of facts is called *complete.* The normal inference processes used in simple MRS applications are correct but not complete.

§4.2 Solving more difficult problems

In chapter 3 we saw how to solve simple problems involving database retrieval. To solve problems whose answers we don't already know, we have to do some inference, for which we need inference rules. Where do we get those from, other than from a book? Recall that our logical connectives are defined in terms of the truth of the propositions they connect — hence the validity of the inference rules for AND given earlier. Rules for the other connectives can be similarly derived; for example

$$\frac{A}{(OR\ A\ B)} \qquad \frac{(OR\ A\ B),\ (NOT\ A)}{B} \qquad \frac{(OR\ A\ B),\ (NOT\ B)}{A}$$

The following is the basic inference rule, called *Modus Ponens*, used in most MRS work:

$$\frac{(IF\ A\ B),\ A}{B}$$

which basically says that if you know that B follows from A, and you know A, then you can deduce B. For example, if we know (IF (CurrentYear 1985) (Age John 22)) and (CurrentYear 1985) then we can conclude (Age John 22) and solve our problem. The reasons for using this inference rule are

1) Most of our knowledge is naturally expressed using IF.
2) Even if it's not, it can usually be rewritten that way.

The truth of these two statements will become apparent.

Let us return to a previous example, the use of (Age $y 22). The diligent reader will no doubt have spotted that this is really a dumb thing to have in the data base. For instance, it implies such things as (Age 22 22). What we should have said was something like (IF (Person $y) (Age $y 22)). Then, given (Person John), we can infer (Age John 22).

Or can we? As it stands, no. The above rule of inference requires that the exact antecedent A be in the data base. A moment's thought, which you should think, yields the extended rule

$$\frac{(IF\ A\ B),\ A'}{B\sigma} \quad \text{where } A\sigma = A'\sigma$$

i.e. if a known fact A' *unifies* with the antecedent A of an IF-proposition, then we can infer the consequent B modified by substitutions from the unifier σ. This may seem highly technical, but really it's just a formalization of your commonsense intuition of how such IF-propositions should be used.

Typically, an IF-proposition, henceforth known as a rule (not to be confused with an inference rule), has a more complex antecedent than an atomic proposition (i.e., a proposition with no connectives), although the conclusion will usually be atomic. To avoid having to have a whole, complex fact stored in the database to unify with the antecedent, which wouldn't be usable except for one particular rule, we must add some inference rules for combining atomic propositions into complex facts:

$$\frac{A_1,\ \ldots,\ A_n}{(AND\ A_1 \ldots A_n)}$$

$$\frac{A_i}{(OR\ A_1 \ldots A_n)} \text{ for } i = 1 \ldots n.$$

Given these inference rules and database such as

```
(IF (AND (Happy $x) (KnowsHappy $x) (HasHands $x $y))
    (ShouldClap $x $y))
(Happy John)
(KnowsHappy John)
(HasHands John JohnsHands)
```

we can deduce

```
(ShouldClap John JchnsHands).
```

§4.3 Solving really very difficult problems indeed

To get all this to hang together, we need a method for performing multiple inferences and stringing them together so that we get from the facts at hand to a solution to the user's query.

Let's start with the facts at hand. Clearly, one way of getting the solution is to find a rule whose antecedent is satisfied, apply the rule of inference and deduce the consequence. We could then add the consequence to the data base and start again, until we deduce a fact that unifies with the query. This is called *forward chaining*, for obvious reasons. The obvious drawback with the scheme is that the system could end up making dozens of inferences that bear no relation to the task of proving the query.

The other simple alternative is to start with the query and ask "How can we prove this?" The answer: prove the antecedent of a rule whose consequence unifies with the query. That is, if we have to prove B, and there is a rule in the data base $(IF\ (AND\ A_1 \ldots A_n)\ B')$ such that $B'\sigma = B\sigma$, then

the task reduces to proving $A_i\sigma$ for all i. The regression ceases when we find an antecedent that's already in the data base. In this case, we only examine inferences that potentially contribute to the actual goal. This is called *backward chaining*. The obvious drawback is that the system might go off trying to prove antecedents that are unprovable, or even false. The majority of expert systems built thus far are basically backward chaining, with some refinements. The "expert knowledge" is encoded as a collection of rules for drawing conclusions under certain circumstances; often, the system has the option of asking the user to confirm an antecedent if it's not in the data base and can't be proven.

The two methods are illustrated in the following sections.

§4.4 Using inference to get results

Before we can do any inference, we'd better have some rules. The following rules define some family relationships (we'll omit MRS's responses to STASH commands from now on):

```
>(STASH '(IF (AND (Parent $p $c) (Female $p)) (Mother $p $c)))
>(STASH '(IF (AND (Parent $p $c) (Male $p)) (Father $p $c)))
>(STASH '(IF (Parent $p $c) (Child $c $p)))
>(STASH '(IF (AND (Child $c $p) (Female $c)) (Daughter $c $p)))
>(STASH '(IF (AND (Child $c $p) (Child $p $g)) (Grandchild $c $g)))
```

Let's also extend our family by giving Bert some kids:

```
>(STASH '(Parent Bert Cathy)
>(STASH '(Female Cathy))
>(STASH '(Parent Bert Chuck)
>(STASH '(Male Chuck))
```

The normal way to solve problems in MRS is to use backward-chaining, the reason being that most data bases are more amenable to this approach; we will see later that the MRS user in fact has a good deal of control over the actual strategy to be adopted. First let's find out who is Bert's daughter; to do this we use TRUEP, which is like LOOKUP except that it uses backward inference as well as data base retrieval to find the answer:

```
>(LOOKUP '(Daughter $d Bert))
NIL
>(TRUEP '(Daughter $d Bert))
(($D . CATHY) (T . T))
```

It is important to understand how TRUEP arrived at its answer. The first step in any attempt to prove a goal is to see if it is already known to be true. Thus TRUEP calls LOOKUPS, which fails. Then TRUEP looks in the data base to find those rules whose consequents unify with the goal. Here there is only one rule

```
(IF (AND (Child $c $p) (Female $c)) (Daughter $c $p)).
```

After applying the unifier to the antecedent, we have the goal

```
(AND (Child $c Bert) (Female $c)).
```

To prove a *conjuctive goal* like this we need to prove all the conjuncts: TRUEP attempts the subgoals from left to right, but this is an arbitrary choice and one which you can alter when you know how. To prove `(Child $c Bert)`, since LOOKUPS fails, we must use the rule

```
(IF (Parent $p $c) (Child $c $p))
```

so we must then prove

```
(Parent Bert $c).
```

LOOKUPS succeeds here, returning to TRUEP a binding list

```
(((Sp . Chuck) (T . T)) ((Sp . Cathy) (T . T))).
```

There are no rules for concluding parenthood, so that's all the solutions there are. Having got these two answers to (Child $c Bert), we must try to prove (Female $c) with each in turn:

```
(Female Chuck)
```

fails because it's not in the data base and there are no rules for concluding such a proposition (at least not in the data base).

```
(Female Cathy)
```

succeeds in LOOKUPS, so

```
(AND (Child $c Bert) (Female $c))
```

succeeds with $c bound to Cathy and the binding list

```
((SD . CATHY) (T . T))
```

is returned after the appropriate substitutions.

Note that if LOOKUP rather than LOOKUPS had been used, TRUEP would not have found the answer since only (Parent Bert Chuck) would have been found. Thus even though TRUEP only needs to return one solution, all alternatives retrieved must be considered. Similarly, in *proving* a goal, we must not ignore any possible solutions. To do this, believe it or not, TRUEP actually calls TRUEPS.

Perform a similar analysis of the proof procedure for the following:

```
>(TRUEPS '(Grandchild $c Alice))
(((SC . CHUCK) ($1 . BERT) (T . T))
 ((SC . CATHY) ($1 . BERT) (T . T)))
>(TRUEPS '($r Alf Bert))
(((SR . PARENT) (T . T)) ((SR . FATHER) (T . T)))
```

Looking at the returned list of Alice's grandchildren, you may be wondering what $1 is doing there. The reason is that sometimes the bindings of *intermediate* variables, i.e. variables that aren't in the query and are unbound when their rules are invoked, are useful in understanding how an answer is arrived at. Here, $1 is the system-created variable that replaces $p (for the purposes of variable standardisation) in the rule defining the Grandchild relation. Thus it informs us that Bert is the parent of Alice's grandchildren and it was this relationship that allowed the system to complete the inference. The rest of the guide will omit these bindings for the sake of clarity; sometimes the intermediate variables are so numerous that binding lists become almost 'illegible'. To overcome this you can process them using getvar and related functions described in chapter 10.

§4.5 Using forward chaining

Forward chaining in MRS is not quite as simple as backward chaining. If the processes were entirely analogous, we would give the system a query then have it reason forward from all the facts in the

database until the solution was produced or until no more inferences could be made. However, this would be somewhat inefficient since most of the inferences would probably be irrelevant. In fact, one full run of such a forward chainer would produce all possible solutions for all possible queries, and can easily take forever.

The method we adopt is to assume that a certain small set of facts constitutes a description of the problem situation. These facts, together with the background knowledge in the database, should contain the seeds of the solution to the query the user has in mind. Thus, when the user adds the problem description to the database, the system forward-chains *from these facts* until no more inferences can be made. Then the user need only do a LOOKUP for her query and *voilà*, the answer is there.

As you may have guessed, the user must add the facts using ASSERT. But first, to notify MRS that we would like to forward-chain from assertions, enter

```
>(STASH '(toassert &p fc)).
```

Don't worry yet about how this works. Let us now redo the example of the previous section. Assuming the rules defining family relations are already in the database, we will assert each fact describing our particular family in turn, and observe the actions of the forward chainer. To do this, we can use the tracing mechanism of MRS to see each step of the inference process as it happens, by entering

```
>(TRACETASK '&x).
```

Work through the following transcript and make sure you see how each conclusion is reached. Each FCDISP step shows a fact being asserted. After it is in the database, the system tries to find all those rules which have a premise, or a conjunct in their premise, that unifies with the fact. For each such rule, it then performs a LOOKUP on each of the other premise conjuncts (if any), and if successful calls FCDISP on the conclusion of the rule.

```
>(ASSERT '(Parent Alice Bert))
Executing FCDISP on   (PARENT ALICE BERT)
Executing FCDISP on   (CHILD BERT ALICE)
DONE
>(ASSERT '(Parent Alf Bert))
Executing FCDISP on   (PARENT ALF BERT)
Executing FCDISP on   (CHILD BERT ALF)
DONE
>(ASSERT '(Female Alice))
Executing FCDISP on   (FEMALE ALICE)
Executing FCDISP on   (MOTHER ALICE BERT)
DONE
>(ASSERT '(Male Alf))
Executing FCDISP on   (MALE ALF)
Executing FCDISP on   (FATHER ALF BERT)
DONE
>(ASSERT '(Male Bert))
Executing FCDISP on   (MALE BERT)
DONE
>(ASSERT '(Parent Bert Cathy))
Executing FCDISP on   (PARENT BERT CATHY)
```

```
Executing FCDISP on   (CHILD CATHY BERT)
Executing FCDISP on   (GRANDCHILD CATHY ALICE)
Executing FCDISP on   (GRANDCHILD CATHY ALF)
Executing FCDISP on   (FATHER BERT CATHY)
DONE
>(ASSERT '(Female Cathy))
Executing FCDISP on   (FEMALE CATHY)
Executing FCDISP on   (DAUGHTER CATHY BERT)
DONE
>(ASSERT '(Parent Bert Chuck))
Executing FCDISP on   (PARENT BERT CHUCK)
Executing FCDISP on   (CHILD CHUCK BERT)
Executing FCDISP on   (GRANDCHILD CHUCK ALICE)
Executing FCDISP on   (GRANDCHILD CHUCK ALF)
Executing FCDISP on   (FATHER BERT CHUCK)
DONE
>(ASSERT '(Male Chuck))
Executing FCDISP on   (MALE CHUCK)
DONE
>(UNTRACETASK)
(&X)
```

After this process, the query (Daughter $d Bert) already has its solution in the database, so a LOOKUP is sufficient to find it.

There are some restrictions on the forward-chaining routine as currently implemented. These mean that the only rules triggered when a fact A is entered will be those of the form (IF A B) or (IF (AND . . A . .) B). Thus instances of the proposition embedded in disjunctions or any other constructions will not be noticed.

§4.6 Solving problems with numbers

MRS knows about certain relations and can ascertain the truth of propositions using them without recourse to the data base. Arithmetic relations are of this type:

```
>(LOOKUP '(> 4 2))
((T . T))
```

MRS knows about > < >= <= * + - //. The latter four are not n-place functions (as in LISP) but $(n+1)$-place relations; for example,

```
(+ $x $y $z)
```

means that $z is the sum of $x and $y.

Thus we can define all kinds of arithmetic relations (not functions) using these as primitives:

```
>(STASH '(IF (AND (+ $x $y $sum) (// $sum 2 $avg))
             (Average $x $y $avg))
>(TRUEP '(Average 7 11 $x))
(($X . 9) (T . T))
```

Note that MRS can only deal with these relations when the arguments are properly bound:

```
>(LOOKUP '(> $x 4))
```

```
NIL
>(LOOKUP '(+ 1 $x 3))
NIL
```

A useful way to view arithmetic and other 'built-in' relations is as *virtual facts*. Apart from the restrictions just noted, we can pretend that the database contains an infinite supply of facts about these relations (their *extensions*, as defined in chapter 1). Thus there are virtual facts like (> 2 1), (+ 5 14 19) and (// 45 7 6) 'available' to LOOKUP. The concept of a virtual fact can be applied to any built-in relation, and several more such relations are given in chapter 8.

You can also use rules with TRUEP to define recursive relations. Although this is a hoary example, it serves to illustrate the technique:

```
>(STASH '(Factorial 0 1))
>(STASH '(IF (AND (> $x 0)
                  (- $x 1 $x-1)
                  (Factorial $x-1 $factx-1)
                  (* $factx-1 $x $factx))
             (Factorial $x $factx))
>(TRUEP '(Factorial 6 $n))
(($N . 720) (T . T))
```

Using the built-in relations can be quite tedious for computing a complex formula since each operator in the formula requires a new conjunct and intermediate variable to hold the result. MRS uses a special relation IS which allows an entire computation to be done in one step with a functional representation taken directly from LISP:

```
>(STASH '(IF (IS (- (* $b $b) (* 4 $a $c)) $d)
             (Discriminant $a $b $c $d)))
>(TRUEP '(Discriminant 2 4 1 $d))
(($D . 8) (T . T))
```

§4.7 Solving problems with lists

A list is an object in the universe just like any other. However, unlike numbers, lists have no ready-made constant symbols which MRS recognises. The one exception is the empty list NIL. Other lists are represented by complex terms. Contrary to the normal syntax for terms, MRS has a special syntax for lists: a list with CAR $x and CDR $y is written ($x . $y). The function symbol '.' appears in the infix position to enhance readability. Other than this, lists are treated the same way as any other terms — it is important to remember that '.' is not a LISP function which is executed, but an uninterpreted symbol.

Let us define the APPEND relation for lists:

```
(APPEND NIL $1 $1)
(IF (APPEND $11 $12 $1)
    (APPEND ($x . $11) $12 ($x . $1)))
```

The recursion works because the empty list NIL can't be unified with the complex term ($x . $1), so TRUEP continues to try the IF-rule until $11 becomes NIL. This may seem a little strange at first, especially if you are used to the CAR and CDR recursion of LISP. Try doing

```
(TRUEP '(APPEND (1 . (2 . (3 . NIL))) (4 . NIL) $1))
```

for yourself on paper to see what happens. Then just to make sure, and to get another insight into why writing facts is better than writing programs, do

```
(TRUEP '(APPEND (1 . NIL) $12 (1 . (2 . (3 . NIL)))))
```

as well.

The use of lists in MRS is far less common than in LISP. The reason for this is that facts usually concern relations between objects rather than between enumerated collections of objects. However, sometimes you will want to know properties of such collections which can only be obtained by examining their contents; for example, the question "How many grandchildren does Alice have?" is asking for the cardinality of the *set* of Alice's grandchildren. MRS provides a built-in relation BAGOF for just this purpose.

```
(BAGOF $x P $s),
```

where P is any proposition involving $x, means that $s is the bag (or *multiset*) containing all $x's satisfying P. The bag itself, to which $s is bound, is just a list term, as in LISP. Since bags (and sets) are represented by ordinary lists they do not have some of the properties of sets one might expect — for example, two sets with the same elements are not necessarily equal, since the elements might appear in different orders.

One of the things about bags is that elements can occur more than once. In some cases these occurrences will be 'spurious' in that we really want the set returned, i.e. the *distinct* solutions for $x of P. This might in fact occur in the case of finding the number of Alice's grandchildren, since there could be multiple ways of showing that someone was related to Alice in this way. For these situations MRS provides a predicate SETOF, which works just like BAGOF except that it removes multiple elements before returning the list. As a result, it is much less efficient than BAGOF and should only be used when necessary.

Before you can use BAGOF or any of the built-in predicates for handling sets you must load the file SET from the MRS directory. Do it now.

OK, now that MRS is apprised of sets, let's try it out:

```
>(LOOKUP '(BAGOF $x (Male $x) $f))
(($F ALF BERT CHUCK) (T . T))
>(TRUEP '(BAGOF $x (Grandchild $x Alf) $g))
(($G CHUCK CATHY) (T . T))
```

Notice that the binding lists for $f and $g look a little odd. If you really believed the story about what MRS lists are, you would expect ($G . (CHUCK . (CATHY . NIL))). But in fact MRS cheats a little and uses the same internal representation as LISP does for lists, with the result that the LISP output routines print out the binding as a normal list structure.

To find out the number of Alice's grandchildren, we just need to find the length of the list representing the bag of them. Thus we need a LENGTH relation; MRS has one built-in, and this is how it's defined:

```
(LENGTH NIL 0)
(IF (AND (LENGTH $l $n)
         (+ $n 1 $nplus1))
    (LENGTH ($x . $l) $nplus1))
```

but the well-known NoOfGrandChildren relation was somehow omitted by MRS's originators so you'll have to put it in.

```
>(STASH '(IF (AND (BAGOF $x (GrandChild $x $y) $g)
                  (LENGTH $g $n))
             (NoOfGrandchildren $y $n)))
>(TRUEP '(NoOfGrandchildren Alf $n))
(($N . 2) (T . T))
```

There are several other built-in relations for handling lists and sets which are described in chapter 8. Using these relations is much more efficient than writing your own, since they use compiled LISP code rather than interpreted MRS facts.

§4.8 Using more complex rules

So far all the rules we have encountered have had a premise consisting of either an atomic proposition or a conjunction of atomic propositions. What of the remaining connectives, OR and NOT? In backward chaining, when a goal (NOT <p>) is encountered, the only way of proving it is to find (NOT <p>) in the database or to find a rule that concludes it — in other words negation is not reducible in the same way as conjunction. On the other hand, disjunction is reducible, since a disjunction of propositions is true if any of the propositions is true. So all we have to do to prove a disjunction is to try proving each disjunct in turn until we find one that is true. As with conjunctions, this is done in left-to-right order. If we have to find all solutions we try all disjuncts. A simple example is the (AbsSign $n $s) predicate, which returns $s=0 for $n=0 and 1 otherwise. One's first, LISP-based instinct is to say

```
(IF (OR (AND (= $n 0) (= $s 0))
        (= $s 1))
    (AbsSign $n $s))
```

which unfortunately doesn't do the right thing at all. The error shows up when we call TRUEPS on AbsSign, which happens when we try to prove the predicate as part of some proof in which AbsSign is embedded. Suppose that, when we get as far as AbsSign, $n is indeed bound to 0, so that AbsSign succeeds with $s bound to 0, but then a later part of the proof fails. MRS will try to find the alternative solutions to earlier parts of the proof to see if, with those solutions, the later part will succeed. Thus it will try the other disjunct (= $s 1) and succeed with that, and carry on the rest of the proof with an incorrect binding for $s. Clearly, the answer is to replace that disjunct with (AND (NOT (= $n 0)) (= $s 1)). But recall that (NOT <p>) can only be proved if a fact tells us that <p> is not the case. It's hardly likely that the database contains facts like (NOT (= 1 0)), (NOT (= 2 0)) and so on. So we are in a quandary. But MRS can help out with a whole new class of connectives called *modal operators*. Whilst a whole body of literature has been written on the semantics of these operators, we will concentrate just on what they mean in terms of the proof process. The operators that MRS provides are KNOWN, UNKNOWN, PROVABLE and UNPROVABLE. Each operates on a single proposition, just like NOT, and means roughly what it says:

(KNOWN <p>)	succeeds if <p> can be satisfied by a simple LOOKUP.
(UNKNOWN <p>)	succeeds if <p> cannot be satisfied by a simple LOOKUP.
(PROVABLE <p>)	succeeds if <p> can be proved from the facts in the database, so it's a null operator.
(UNPROVABLE <p>)	succeeds if <p> cannot be proved from the facts in the database.

In this case, since = is handled using LOOKUP, we should say

```
(IF (OR (AND (= $n 0) (= $s 0))
        (AND (UNKNOWN (= $n 0)) (= $s 1))
    (AbsSign $n $s))
```

UNKNOWN and UNPROVABLE are extrememly useful in all kinds of situations. In any instance where something that isn't known to be true can be assumed to be false, we can use these operators and avoid the chore of having to explicitly stash the negated propositions which the use of NOT would require. This assumption is called the *closed-world assumption*, and is used all the time by us humans. For instance, if I can't see a wall in front of me as I walk down a corridor I tend to assume there isn't one there. In a logic programming environment, we have to make these assumptions a little more explicit, as you will see when doing the exercises in the next chapter.

Chapter 5
Some (almost) real examples

There are (at least) two distinct styles of using MRS corresponding to the situations in which the user finds herself: she may already know how to solve the problem, i.e. have the course of the necessary computation already mapped out; or she may not. The 'code' produced in the two cases is not necessarily dissimilar, in fact one could imagine cases where the same programs were produced by two programmers even though their approaches were totally different. This distinction is reminiscent of that between AI and non-AI programs.

In the first case, where the user knows what computation is needed, facts entered as rules of the form

(IF (AND $A_1 \ldots A_n$) B)

are understood *procedurally* to mean "To prove B, prove A_1 through A_n in that order", assuming backward-chaining is being used. The user thus breaks down her goals into subgoals, often using the results of subgoal A_i in subgoal A_{i+1}, until trivial subgoals are reached. This results in programs that look very much like their LISP equivalents (cf. the definitions of APPEND and Factorial); the MRS user has the additional advantages of the implicit computation in unification and the non-determinism achieved using free variables.

In the second case, where the desired computation is not known, MRS (or at least logic programming) really comes into its own. In the following example we will produce a system that can predict the outputs of an electronic device, consisting of wires and gates, given its inputs. The stages in the general method are as follows:

1) Decide on an *ontology* for the domain — the contents of the universe and their categories.
2) Decide on a *vocabulary* of relations for describing both the problem instances and the general knowledge used in solving problems.
3) Collect and encode all the general knowledge.
4) Encode the description of the particular instance.
5) Invoke the appropriate MRS inference procedure to produce the solution.

Of course, the first three stages are somewhat interdependent: the ontology may depend on the knowledge available; a new problem instance may turn up objects not yet accounted for, and so on. For example, if I have no idea how temperature affects electronic devices I won't want to include temperature in my ontology or relations. Similarly, one often finds the need to rethink one's ontology when one finds that the knowledge is difficult to express in terms of the current set of objects. Thus for some purposes the best order may be somewhat different from that given above.

Rather than just dump the solution on you, let's try to follow the stages leading to it in detail and motivate the decisions leading to the final program.

§5.1 Deciding on an ontology

This is not the same as having a clear definition of the problem. A clear definition of this problem class is that it consists of arbitrary circuits constructed from wires and two-input AND, OR and XOR gates and single-input NOT gates. The terminals of the circuit will be designated as input or output terminals. The full-adder circuit in Fig. 1 is the particular example that we will consider. The first two inputs are the two bits to be added, the third is the carry bit from the previous addition. The first output is the sum bit, the second the carry bit to be included in the next addition.

Presumably we will want to include the gates themselves in our universe since we have to describe their behavior. Similarly the full adder itself has terminals and a behavior (which we are trying to deduce) so we'll include it in the universe and call it F. Since we won't want to describe

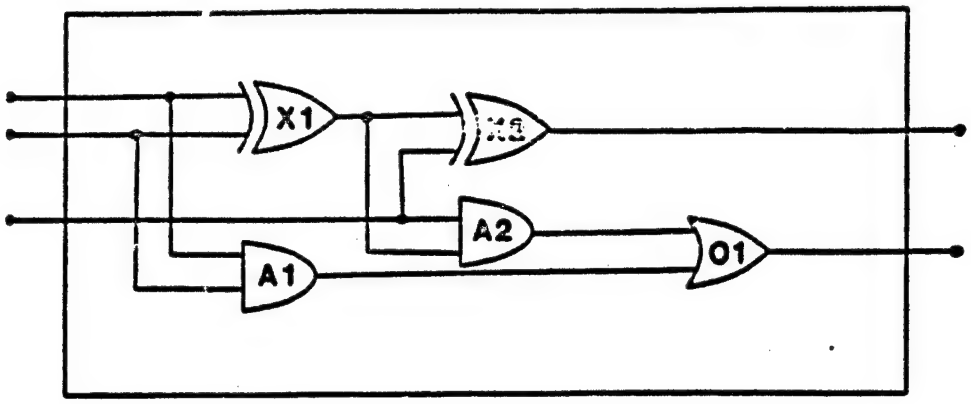


Figure 1. A full adder circuit.

the behavior of each individual gate when we need only describe each type, we'll want the gate types ANDGate, ORGate, XORGate. What else? Wires? Junctions? Terminals? Well, the terminals need to be there because we need to know the i/o signals at them. But the behavior of a circuit in this idealisation is determined by the components and their interconnections regardless of the path or type of those interconnections, so the junctions and wires themselves are irrelevant. Only the fact of the interconnection need be recorded. Which leads naturally to the next stage.

§5.2 Deciding on a vocabulary

First, the description of the individual gates:

```
(Type <gate> <gate-type>) e.g.  (Type A1 ANDGate)
```

Now, the behavior of the devices will be specified in terms of signals on their terminals

```
(Signal <terminal> <value>)
```

where `<value>` will be on or off. We could equally well say

```
(On <terminal>) and (Off <terminal>)
```

but *reifying* the signal will probably allow greater flexibility if needed. For instance, we'll need to say somewhere that the signals at both ends of a wire should be the same; then we can say

```
(IF (AND (Connected $t1 $t2) (Signal $t1 $s))
    (Signal $t2 $s))
```

but with ON and OFF predicates we would have to say

```
(IF (AND (Connected $t1 $t2) (On $t1))
    (On $t2))
(IF (AND (Connected $t1 $t2) (Off $t1))
    (Off $t2))
```

Note that

```
(IF (AND (Connected $t1 $t2) ($s $t1))
    ($s $t2))
```

is no good because it would imply that the terminals were the same color too.

At first sight one might think that the terminals can be named just like the gates: A1Input1, A1Input2 etc. But the rules for gate behavior need to be written for a general gate not just a particular individual, so we need a function symbol which will refer to the general gate's terminals: (Input1 A1) perhaps; if we follow the reification principle then (Input 1 A1) is probably better. So to state that the second input of gate X1 was on, we would say

```
(Signal (Input 2 X1) on).
```

The interconnections can now be specified easily; for example

```
(Connected (Output 1 X1) (Input 1 X2)).
```

§5.3 Collect and encode all the general knowledge

The problem here is to predict the behavior of a device; to solve it we need to know how the signals are propagated. The signals are propagated through wires and gates. The 'wires' are easy to deal with, as shown above:

```
(IF (AND (Connected $t1 $t2) (Signal $t1 $s))
    (Signal $t2 $s))
```

Of course, if we were dealing with time-dependent signals and finite-length wires, or wires with impedance, then the system would need a lot more information.

The propagation of signals through a gate depends on the gate type. The following facts describe the operation of the three types of gate used.

```
(IF (AND (Type $g ANDGate)
         (Signal (Input 1 $g) on)
         (Signal (Input 2 $g) on))
    (Signal (Output 1 $g) on))
(IF (AND (Type $g ANDGate)
         (Signal (Input $n $g) off))
    (Signal (Output 1 $g) off))

(IF (AND (Type $g ORGate)
         (Signal (Input 1 $g) off)
         (Signal (Input 2 $g) off))
    (Signal (Output 1 $g) off))
(IF (AND (Type $g ORGate)
         (Signal (Input $n $g) on))
    (Signal (Output 1 $g) on))

(IF (AND (Type $g XORGate)
         (Signal (Input 1 $g) $i1)
         (Signal (Input 2 $g) $i1))
    (Signal (Output 1 $g) off))
(IF (AND (Type $g XORGate)
         (Signal (Input 1 $g) $i1)
```

```
        (Signal (Input 2 $g) $i2)
        (UNKNOWN (= $i1 $i2)))
    (Signal (Output 1 $g) on))
```

§5.4 Encode the description of the particular instance

In this case the problem instance is two-fold: first the circuit, then the particular inputs. The circuit is described by listing the types of the gates and their interconnections.

```
(Type X1 XORGate)
(Type X2 XORGate)
(Type A1 ANDGate)
(Type A1 ANDGate)
(Type O1 ORGate)
(Connected (Input 1 F) (Input 1 X1))
(Connected (Input 1 F) (Input 1 A1))
(Connected (Input 2 F) (Input 2 X1))
(Connected (Input 2 F) (Input 2 A2))
(Connected (Input 3 F) (Input 2 X2))
(Connected (Input 3 F) (Input 1 A2))
(Connected (Output 1 X1) (Input 1 X2))
(Connected (Output 1 X1) (Input 2 A2))
(Connected (Output 1 A2) (Input 1 O1))
(Connected (Output 1 A1) (Input 2 O1))
(Connected (Output 1 X2) (Output 1 F))
(Connected (Output 1 O1) (Output 2 F))
```

whilst the inputs are specified by giving the signal value at each of the input terminals of the adder:

```
(Signal (Input 1 F) on)
(Signal (Input 2 F) off)
(Signal (Input 3 F) on)
```

§5.5 Invoke the appropriate MRS inference procedure

To check that the circuit does what we want, we need to check both outputs. A TRUEP for each would suffice, but we can use the power of indeterminacy to get MRS to go through all the outputs itself.

```
>(TRUEPS '(Signal (Output $n F) $s))
((($N . 1) ($S . OFF) (T . T))
 (($N . 2) ($S . ON) (T . T)))
```

which is the correct answer.

You may say "That's all very well for those inputs, but what about all the other combinations?" It would certainly take a lot of boring typing to stash and then unstash all eight combinations of the three inputs. But we can get MRS to enumerate them, given a bit of thought. To find the possible inputs, it needs to know the possible values for the signal on a terminal. At the moment, they could be on, off, green or angry for all it knows. So

```
(SignalValue on)
```

```
(SignalValue off)
```

tell it what it needs to know. Then if we define a predicate which is satisfied by any combination of inputs with their respective outputs and call TRUEPS on it, MRS will go through all possible inputs for us.

```
(IF (AND (SignalValue $i1) (Signal (Input 1 F) $i1)
         (SignalValue $i2) (Signal (Input 2 F) $i2)
         (SignalValue $i3) (Signal (Input 3 F) $i3)
         (Signal (Output $n F) $s))
    (InputTested $n $i1 $i2 $i3 $s))
>(TRUEPS '(InputTested $n $i1 $i2 $i3 $s))
((($N . 1) ($I1 . OFF) ($I2 . OFF) ($I3 . OFF) ($S . OFF) (T . T)) etc.
```

The kind of reasoning by which the above answers are produced is extremely important and forms the basis of all scientific thought from the time of Newton up to the advent of quantum mechanics. Basically it relies on the notion that, given a description of the initial situation and some correct laws on how one situation follows from a preceding one, the situation at any future time can be predicted. The knowledge base and case description are said to form a *causal model* of the system; such models are increasingly being employed in expert systems that deal with physical situations.

You may have a nagging intuition that it's an odd thing to do to work back from the outputs when the 'flow of causality' starts from the inputs. This intuition is well-grounded — a forward chaining approach would be more efficient since all the inferences would be necessary and determined, whilst the backward chainer may be trying to prove output values that are inconsistent with the inputs before it makes the correct choice. A highly instructive exercise is to try it both ways with tracing turned on.

§5.6 Exercises

The trouble with exercises is that people are just too fat and lazy to do them. You won't lose weight by doing these, in fact you could go jogging instead, but they will test and extend your ability and understanding, I hope.

1. Write some rules to play a move in tictactoe. The board representation will be

```
(On <player> <square>)
```

where the player is O or X and the squares are numbered 1...9 left to right, top to bottom. To produce a move the user should be able to type simply

```
>(TRUEP '(BestMove X $move))
((/$MOVE . 5) (T . T))
```

Use only rudimentary strategy: take a win if available; stop an opponent's win if necessary; move at random otherwise.

2. If you thought that was a little too easy, now do the same for a chess move. Include as many details of castling, pawn promotion, *en passant* moves and checking as possible. To make things a little easier, you don't need to pick the best move; just do enough so that

```
(TRUEPS '(LegalMove White $move))
```

would return all of White's legal moves. You will have to decide how to represent the board and the moves; also some history of the game will have to be present to decide on castling and *en passant* legality.

Notice that by writing rules that decide if an individual move is legal you have defined the space of all legal moves and your rules can be used as a *generator* as well as a tester.

3. Create a knowledge base and problem description sufficient to solve the geometrical problem presented in Fig. 2.

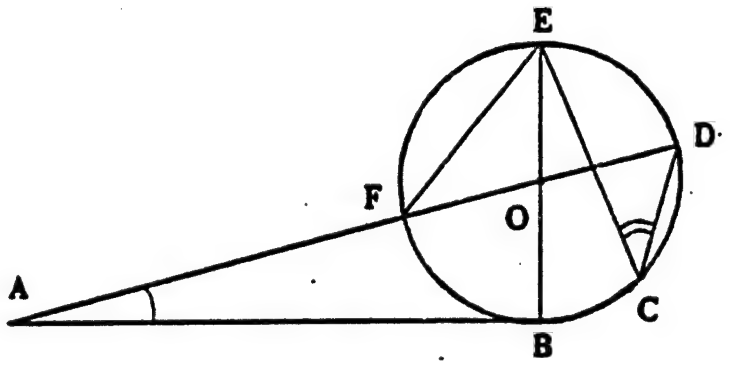


Figure 2. Given $<OAB = 20°$ find $<ECD$

Try to follow the stages outlined earlier in the chapter for the circuit example. A hint to save you some head-scratching: the easiest way to say that AB is a tangent to the circle is to say that AB is a tangent to the circle.

Chapter 6

Controlling deduction

So far our description of MRS has concentrated on the solving of problems at the domain level using mainly backward-chaining inference. In this capacity, MRS is little different from other systems such as PROLOG. The real distinction lies in MRS's ability to allow the user to express all the knowledge she has about the best way to go about finding the solution, efficient ways of doing particular computations or the overall structure of the computations she would like to see performed (the *architecture*). Essentially, at each stage of its operation MRS uses a theorem-prover to find out how to proceed; by making facts available to this theorem-prover the user can tell MRS what to do and how and when to do it.

§6.1 Tasks

From the point of view of the actions that are being performed, computation in MRS consists of the creation and execution of *tasks*. Tasks are calls to LISP subroutines with their arguments; they range from calls to proof routines such as TRUEP through single proof steps to output routine calls. Executing one task can cause other tasks to be created. Given a method for making tasks available for execution, a method for finding out how to perform the tasks and a method for deciding which of several tasks to execute, we have a general architecture for computation capable of producing any desired behavior.

§6.2 Controlling what gets done when: the scheduler

The question of *what* gets done requires a discussion of the mechanics of the top level of MRS — the scheduler routine.

6.2.1 The general (non-default) mode of scheduler operation

The scheduler is invoked by all the built-in inference routines. In its most general mode of operation it follows the deliberation-action model of intelligent systems shown in Figure 4. To get the scheduler to operate in this mode the switches executable and executed must be set to T.

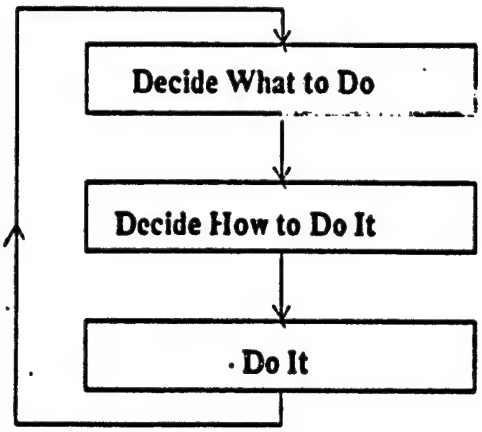


Figure 4 The deliberation-action model

The deliberation part is achieved by having the meta-level theorem-prover find a task satisfying (executable &task). There are of course meta-level rules to decide what tasks are executable:

```
(IF (AND (applicable &task)
         (UNPROVABLE (disqualified &task)))
    (executable &task))
```

A task is disqualified if another applicable task is preferred to it:

```
(IF (AND (applicable &anytask)
         (preferred &anytask &task))
    (disqualified &task))
```

To find the applicable tasks, we first find the runnable tasks; there are no built-in rules for deducing runnability so this is where the user can decide what gets done. All runnable tasks with operators which are LISP subroutines are automatically applicable.

Another switch **preferred** determines whether tasks can be disqualified — if it is NIL the above rule for proving **disqualified** will not be used, whereas if it is non-NIL the rule will be in effect.

A simple example of how we can plug into the deliberation process to affect what gets done is the implementation of *demons*. A demon is a task that is to be executed whenever a given set of conditions becomes true. We can signal that a task should be executed (or at least scheduled for execution) by asserting that it is runnable. This can be done automatically if we use forward chaining from assertions and express the demon as a rule of the form

```
(IF <triggering condition>
    (runnable <task to be executed>)).
```

Using this mechanism, the range of system architectures that can be implemented is enlarged to include blackboard systems, something one wouldn't expect from a PROLOG look-alike. Essentially, these demons form a *condition-action* architecture, which can be used to implement any desired structure of computation.

6.2.2 Default mode — the agenda

The above set of rules is only used in full when the switches **executable** and **executed** are non-null. The normal mode of operation is based on an *agenda*. The variable agenda stores a list of tasks, all of which are automatically applicable. If the switch **preferred** is NIL, the default, the executable task is the first one on the agenda. Thus in the default mode the agenda is empty until a top-level command is entered by the user. The LISP function (truep or whatever) that is invoked places itself and its arguments on the agenda and calls **scheduler**, thus connecting with the scheme described above. All the built-in inference routines use the agenda; the normal base-level backward-chainer **bc** puts **bcdisp** tasks on the agenda; **fc** puts **fcdisp** tasks on the agenda, and so on.

When the **preferred** flag is non-null, the tasks on the agenda are compared to find the most preferable one, thus disqualifying all the others, at least in theory. In practice, for efficiency reasons, the **executable**-related rules are skipped, and the preferred task is executed immediately.

§6.3 Telling MRS how to do things

This section introduces the ideas involved in specifying how MRS is to execute the tasks it encounters. The range of tasks which can be handled is given in the section on task-related attachment in chapter 8; here we present some motivation and a detailed example of the kind of information the user can give the system for deciding how to perform tasks.

We have already seen some strange mumbling necessary for doing forward chaining. The exact incantation was

```
(STASH '(toassert &p fc)).
```

Its effect was that all subsequent assertions caused forward-chaining to take place. Instead of using the normal LISP subroutine for assertions (which is the same as that for STASH initially), MRS will call fc with the asserted proposition as argument. Whenever a task such as an ASSERT or TRUEP is scheduled to be executed, MRS looks up the appropriate subroutine to use under toassert or totruep. Thus by stashing facts like (toassert &x fc) we can affect the way in which MRS performs the commands we give it. Such facts are qualitatively different from the domain facts since they deal with how those facts are to be used rather than stating truths about objects in the universe; they refer to the binding status of variables, the order of processing of conjuncts, the method of representation for fact classes rather than Zen or automotive diagnosis and repair. Thus they encode *meta-knowledge* (meaning "knowledge about knowledge") and are said to be at the *meta-level*. It is MRS's extensive facilities for representing and using this kind of information that give it the name "Meta-level Representation System". We will see many more examples of meta-level knowledge, but first we need some background to show why it's necessary.

We have already seen, in the case of predicting the outputs of electronic devices, that forward-chaining can be more efficient than backward chaining. This piece of meta-level information was put into action by asserting the input values rather than stashing them and by telling MRS to forward-chain from assertions. When is this a good idea in general? To discuss this, we need to think about the *structure* of the rule base. Suppose that, for any given conclusion (such as the value of an output), there a dozen different rules for deducing it. Then a backward-chaining system has to try all of these even if only one of them leads to a solution. Then again, if a given fact (say the value of an input) is used in the premises of a dozen different rules, then a forward-chaining system might trigger all of these rules when only one of them leads to the desired answer. Fig. 3 illustrates the difference in the two types of rule base in diagrams showing rules as nodes, with an arc showing that the conclusion of the rule at the left end unifies with part of the premise of the rule at the right end.

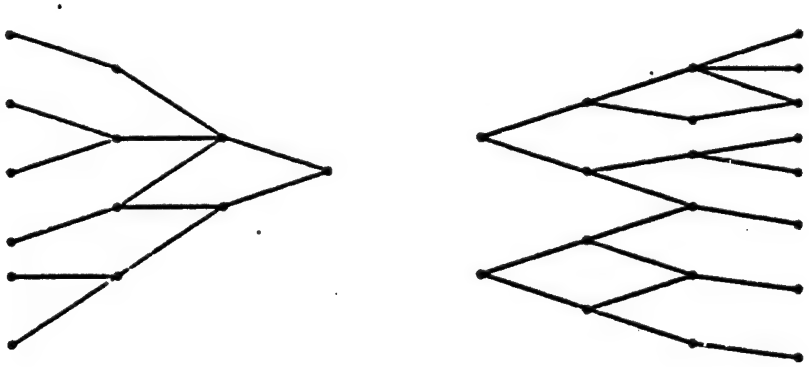


Figure 3 Good for forward chaining Good for backward chaining

Applying this simple analysis to the family relations example in section 4.2, we see that the only chaining occurs on the Child relation; it appears in one conclusion but three premises, therefore the database structure is best suited for backward-chaining.

Certainly it's true that in many real cases the choice is not so simple, for example when the database diagram contains loops. But we shall now try to get MRS to do this kind of analysis automatically. Being a meta-level system, MRS can *reason* about how to do things using facts which the user can provide, thereby directly influencing the operation of the system. Specifically, it uses a backward-chaining theorem-prover and meta-level facts to solve meta-level problems. When given a command such (ASSERT <p>) MRS calls a stripped-down version of TRUEP called trtruep on the goal (toassert <p> &method). trtruep is stripped-down in the sense that it doesn't have access to any meta-meta-level, so it runs pure backward-chaining. Thus all that's necessary is to stash a fact of the form

```
(IF <database suitable for forward-chaining>
    (toassert &x fc)).
```

Notice that, at the meta-level, variables begin with & instead of $. The meta-level theorem-prover treats all base-level variables as constants; the base-level theorem prover treats meta-level variables as constants. This is actually done by having two different unification routines, one for each level.

Often, users who are happy with base-level programming are wary of working at the meta-level, perhaps equating it with 'system hacking'. Nothing could be further from the truth: the meta-level is for expressing and using abstract, high-level knowledge about how problems should be solved. So to overcome your trepidation or distaste, we're going to plunge in and do an example that's probably more complex than anything you'll ever want to do at the meta-level. We shall implement a simple definition of the predicate

```
<database suitable for forward chaining>
```

which will involve some quite tricky problems.

Interpreting simplistically the above analysis, we'll say that a database is suitable for forward chaining if

the average number of premises unifiable with each conclusion
    is less than
the average number of conclusions unifiable with each premise.

This approximately corresponds to the forward-search branching factor being smaller than that for backward search.

To convert this into an MRS predicate, the top-down approach will be used. The basic condition is

```
(IF (FC-Indicated) (toassert &p fc)).
```

Note that FC-Indicated has no arguments: it is a condition on the whole database available at the time we make the assertion that causes MRS to try to find out how toassert. From the above definition we have

```
(IF (AND (ForwardBranch &ffactor)
         (BackwardBranch &bfactor)
         (< &ffactor &bfactor))
    (FC-Indicated))
```

We will define just the forward branching factor here and leave the rest to the reader's fertile

imagination.

```
(IF (AND (BAGOF &n
                (AND (IF &prem &concl)
                     (NoOfMatchingPremises &concl &n))
                &matchnumbers)
         (Average &matchnumbers &ffactor))
    (ForwardBranch &ffactor))
```

To understand this, recall that (BAGOF <X> <P> <S>) tries to find every solution for <P>. Thus for the first part of the conjunction, (IF &prem &concl), what actually happens is that LOOKUPS returns all the rules in the database just as if IF were an ordinary predicate, and &prem and &concl are bound appropriately. Then for each &concl we find the number of premises it matches (which is the number of rules it can trigger in forward chaining) and BAGOF returns a list of these numbers.

Carefully avoiding insulting the reader with an exposition of averaging, the only remaining problem is NoOfMatchingPremises. The approach is similar to the outer loop: make a bag of all the rules in the database, then find the length of the bag; but this time only include those rules with a premise that matches the &concl we're looking at.

```
(IF (AND (BAGOF &lhs    ;lhs is as good as the rule if we're only counting.
                (AND (IF &lhs &rhs)
                     (MatchingPremise &concl &lhs))
                &matchingrules)
         (LENGTH &matchingrules &n))
    (NoOfMatchingPremises &concl &n))
```

To write MatchingPremise, we have to deal with the two basic types of premise — the atomic proposition and the conjunction of atomic propositions. We'll ignore disjunctions and more complex forms for now.

Whatever the type, we must have a way of deciding if two propositions match. We can't use = because we are comparing base-level propositions and to prove = the meta-level theorem-prover uses the meta-level unifier. There is a base-level unification routine available called batchp, but it's a LISP function not a built-in mrs predicate. So there two questions that spring to mind: firstly, how to interface to a LISP subroutine so that it looks like an MRS predicate; secondly, how to discover that batchp is the name of the function we need. The answer to the second question is that chapter 10 deals with all the system routines available that might be useful to the user. The first question is more tricky.

Normally, to interface a LISP routine to make it look like a predicate you would use the "computable representation" mechanism or stash a procedural attachment for it using totruep (both discussed in chapter 8).

However, the meta-level theorem-prover is a stripped-down version that doesn't cater for these luxuries. A cheap and cheerful method that works at the meta-level as well as the base level is to use the Done built-in predicate:

```
(Done <LISP expression> <term>)
```

succeeds if the result returned from the execution of the LISP expression unifies with the term. Obviously, any MRS variables in the expression will be instantiated (if possible) before the call to LISP is made. In this case, we know that batchp succeeds if it returns a list (as opposed to NIL) so

we just have to make sure the term only unifies with a list. After this horrendous plunge into detail, we are ready to write MatchingPremise.

```
(IF (AND (= &lhs (AND . &premises))
         (Member &premise &premises)
         (Done (Batchp &premise &concl) (&bindings)))
    (MatchingPremise &concl &lhs))
(IF (AND (UNKNOWN (= &lhs (AND . &premises)))
         (Done (Batchp &lhs &concl) (&bindings)))
    (MatchingPremise &concl &lhs))
```

The purpose of going through this example in gory detail has been not so much to provide a useful meta-level tool for database analysis (it would be quite inefficient to go through this analysis for every ASSERT), but more to show that programming at the meta-level is no harder than at the base level, or perhaps I should say just as easy. The difference is simply that the subjects of the meta-level predicates are facts instead of domain objects.

§6.4 Expressing control strategies at the meta-level

In most programming languages, there are instructions that achieve the necessary computations and there are instructions that order those computations, decide which to perform and how often, and in general decide what gets done. The instructions look just like the computational instructions, use data structures such as flags, index registers, queues and so on, and are generally intermingled with and seldom distinguished from the rest of the program.

We have already seen that in MRS the concept of a program as a series of instructions is replaced by the complementary ideas of knowledge and inference. The same process can be applied to the control structure of a program. The control structure is really a procedural expression of meta-level knowledge about what should be done when. The natural course of action in MRS is to express this knowledge *explicitly* and use it inferentially to decide what to do. In default mode, MRS just uses depth-first backward chaining, using facts and rules in order of recency of creation, and proving conjuncts and disjuncts left to right. These are all arbitrary choices. Meta-level control knowledge is expressed by specifying *preferences* between tasks as to which should be done first, and this is sufficient to allow a broad range of control strategies to be implemented.

The lowest level of task is the single inference step. The task preferences are expressed using the predicate PREFERRED; thus when the system has a choice of tasks, as when more than one rule can be used to prove a proposition, it tries, for each pair of pending tasks, to prove the proposition

```
(PREFERRED &task1 &task2).
```

The preference relation induces a partial ordering on the tasks, and the most preferred task is chosen for execution. However, since this mechanism is time-consuming and not always needed, the default mode of operation ignores preferences. The **preferred** flag should be set to T for this facility to operate.

By using conditional preferences, which depend on arbitrarily complex task properties, we can create very sophisticated control structures. One limitation on the complexity is the amount of information available in the task description; the information can include the context in which the task is being invoked, the history of the computation leading to the invocation, the available resources for its completion, the reasons for its invocation and so on. These considerations are particularly important in constructing autonomous systems and new inference routines. The following section deals with a concrete task class.

§6.5 Control structure examples in backward chaining

Since backward chaining is the usual theorem-proving method at the base level, we will show in detail how task ordering can be used to implement control structures for it in a number of ways. The routine that performs the single inference step is bcdisp, which has four arguments:

gl	the list of goals to be satisfied.
al	the binding list for the variables in gl.
jl	a justification list containing the names (P123 etc.) of the facts used in deriving the current gl from its predecessor.
ce	a stack of the goal lists preceding the current gl. Each list is followed by its corresponding jl.

Thus, if we had a query Q, and a database containing

```
P112:  (IF (AND A B C) Q)
P113:  A
```

and the system had reached the goal of proving B, the call to bcdisp would have the following arguments:

gl	(B C)
al	the appropriate binding list
jl	(P113)
ce	((A B C) NIL ((AND A B C)) (P112) (Q) NIL)

As you can see, it takes a step to go from a goal list ((AND A B C)) to the subsequent list (A B C).

Now, given two bcdisp tasks

```
(bcdisp &gl1 &al1 &jl1 &ce1)
(bcdisp &gl2 &al2 &jl2 &ce2),
```

how do we express control knowledge as a preference between them? Obviously we will use the meta-level inference capability of MRS to make the choice dependent on some condition on the two sets of arguments:

```
(IF <condition on both sets of gl, al, jl and ce>
    (PREFERRED (bcdisp &gl1 &al1 &jl1 &ce1)
               (bcdisp &gl2 &al2 &jl2 &ce2)))
```

Let us view the theorem-proving process as a search. If we wanted to implement a breadth-first architecture, rather than the default depth-first (e.g. if we wanted to guarantee finding the shortest proof) we would simply use the condition

```
(IF (AND (LENGTH &ce1 &l1)
         (LENGTH &ce2 &l2)
         (< &l1 &l2))
    (PREFERRED (bcdisp &gl1 &al1 &jl1 &ce1)
               (bcdisp &gl2 &al2 &jl2 &ce2)))
```

since this means that the shortest existing derivation path will always be expanded first.

If we wrote the length condition on gl instead of ce we would have a simple best-first search, based on the shaky but often useful premise that a smaller number of goals means a speedier solution.

These are the most general forms of preference imaginable. We can use more delicate instruments; for instance, suppose we have a search-based problem-solver which uses a predicate (Succ $parent $child) to generate successors. Then we can implement an evaluation-based best-first search as follows:

```
(IF (AND (Evaluation &parent1 &v1)
         (Evaluation &parent2 &v2)
         (> &v1 &v2))
    (PREFERRED (bcdisp ((Succ &parent1 &child1) .  &gl1) &al1 &jl1 &ce1)
               (bcdisp ((Succ &parent2 &child2) .  &gl2) &al2 &jl2 &ce2)))
```

where presumably Evaluation would, for efficiency purposes, be a procedurally attached LISP function.

Much work remains to be done on the meta-level expression of control strategies, but MRS's capabilities are sufficiently general that the user should be able to devise a way to express cleanly the control knowledge that she has.

Part II :

Using the advanced features of MRS

Chapter 7

Theories: individually wrapped databases

The word *theory* in MRS is used to describe a set of facts in their own database. Up to now, we have treated all facts as equals, belonging in one big, finite but unbounded database. With the notion of theories we can begin to treat databases as objects in themselves. One of the things objects often have is a name. The default database you have been using so far is called the **global** theory. All the facts in it know they're there: the proposition symbol for each fact has on its **theory** property the word **global**.

Clearly, the burning question is "Why would I want to have more than one theory?". Well, the technical use of the term is not so far from the everyday meaning; if you have more than one theory about something then you can use more than one theory to keep your theories in. The competing theories might be as different as the wave and particle theories of light, or the Freudian and Gestalt theories of human behavior; or they might just describe different hypothetical situations in a search space. Another common use of theories is for efficiency purposes. If one is solving mathematical puzzles then there is no need to search for rules and facts through an encyclopædic collection of knowledge about the lives of composers and ecclesiastical architecture that might be present in a global theory. By dividing the total knowledge into different areas, one achieves an automatic focusing of attention onto the relevant information if the appropriate theories are used.

The key idea is that the ability to treat sets of facts as objects gives the ability to compare, select, rank, exclude, divide, combine, distinguish and otherwise mess around with bodies of knowledge, thus conferring upon the user a rich, new opportunity for the manipulation of information.

§7.1 Getting facts into and out of theories

As mentioned above, all facts stashed by the user go by default into the theory **global**. This is because **global** is the initial value of the variable **theory**, which determines the current default theory for stashing. Thus to create a new theory one can set the value of **theory** and start stashing. A alternative to use the following theory-specific versions of the standard database routines:

thassert	(**thassert** <p> <th>) asserts <p> in theory <th> and sets the value of **theory** to <th>.
thunassert	(**thunassert** <p> <th>) unasserts <p> from theory <th> and sets the value of **theory** to <th>.
thstash	Rather like **thassert**.
thunstash	Rather like **thunassert**.

A theory can be emptied by calling **empty** on it. One can also create a whole theory at one go by saying

(**deftheory** <th> $<p_1>$ ... $<p_n>$)

which first empties <th> and then asserts the propositions into it.

§7.2 Selecting which theories to use

Having a current default theory for stashing is all very well but when it comes to doing a **lookup** one might want to have more than one theory accessible. The value of the variable **activetheories** is a list of the theories which are available to **lookup**. The global theory is always available. Thus you can set the value of **activetheories** yourself or use the commands

(activate <th_1> ... <th_n>)

which adds the specified theories to the active list, and

(deactivate <th_1> ... <th_n>)

which takes them off the active list.

The user can specify a representation for a class of propositions that is specific to a given theory by asserting

(threpn <p> <rpn> <th>).

This will be effective while <th> is active. One should be aware that the result of a lookup on a fact which has two different representations in two active theories is undefined.

The subroutine (contents <th>) prints out a list of the pr facts in a theory, each preceded by its associated proposition symbol.

§7.3 Related theories

Suppose we have a general theory LogicProgramming and some specific theories such as MRSProgramming and PROLOGProgramming. All of the facts in the specific theories are notionally part of the subject matter of logic programmig; thus when we activate LogicProgramming we would like the language-specific facts to be available also without having to duplicate them in the overall theory or activate each subtheory explicitly. MRS allows the user to do just this (surprise, surprise) by simply asserting

(includes 'LogicProgramming 'MRSProgramming)

and so on. The effects can be undone by asserting an unincludes fact for the pair of theories. includes and unincludes are also available as subroutines which can be called with the theories as arguments, giving greater efficiency.

Chapter 8

Procedural attachment

Procedural attachment is a term which denotes the interfacing of procedural information (i.e. applicative or imperative code) to a declarative system. The purpose is to achieve greater efficiency for certain operations at the expense of the generality and explicitness provided by the mechanisms of deduction.

Procedural attachments in MRS come in two flavours: the first type might be called task-related and involves the replacement of system functions for say proof or retrieval with special-purpose user code or other, non-standard routines; the second type is predicate-related, involving replacing the normal deductive or look-up procedures for certain predicates with programs that achieve the same end with greater speed or using less space.

§8.1 Task-related attachment

For any given top-level system task <T>, the user can designate the LISP function to perform it for arguments matching a pattern by stashing a fact of the form

```
(to<T> <pattern> <function name>).
```

As previously mentioned, after the task is invoked by the user MRS will attempt to find out how to perform it by looking for just such `to<T>` facts in the database. A precautionary note: since only one way of performing a task is needed, MRS will just use the first one it finds; the fact that the meta-level theorem prover does a lookup before trying rules means that unconditional propositions will always have precedence over rules, so default procedures may have to be unstashed before conditional attachments become effective. For instance, the default assertion method is stored as

```
(toassert &p pr-stash)
```

so if you wanted to add a conditional fact such as

```
(IF (FC-Desirable &p) (toassert &p fc))
```

you should first either remove the default by unstashing it, or replace it with a conditional default whose antecedent is always true (the standard one is `(= T T)`).

The tasks for which this mechanism is implemented are as follows:

`(un)assert`	Usual choice is whether to forward-chain or not. Certain 'system' facts require special routines, e.g. `(toassert (repn . &x) repn-assert)`. Each representation method also has its own routine.
`lookup(s)`	Representation-dependent.
`(un)stash`	Representation-dependent.
`truep(s)`	To procedurally attach particular predicates or change the inference method.
`cache`	Special case — invoked automatically but otherwise like `stash`. See chapter 11 for a discussion of caching.
`edit`	To specify the editor to be used for direct database editing (see chapter 12).
`monitor(s)`	To affect how database assertions are monitored (see chapter 12).

`output(s)`	To affect how facts are displayed (see chapter 12).

Not all of these tasks can be handled independently. If a fact is stashed using a non-standard representation, it must be retrieved using the appropriate routine for that representation. Such co-ordinated changes are best handled using the repn mechanism described in the next chapter. In that chapter we also deal with the alternative inference routines available, some of which require specialised representations also.

§8.2 Predicate-related attachment — built-in predicates

As we have already seen, some predicates in MRS are evaluable; that is, they are not defined by rules or simple facts, but have computable truth conditions. The arithmetic relations are the prime examples. Equality may also be viewed as a computable predicate, with a procedural attachment to the unification routine. The predicates for handling sets and lists, including `length`, are also evaluable.

These predicates are all attached using `tolookup(s)` and `totruep(s)` facts which are in the initial MRS database. You can inspect these facts by calling `PRFACTS` on the predicate in question.

`*`	(* $x_1 \ldots x_n$ y) succeeds if y unifies with the product of $x_1 \ldots x_n$.
`+`	(+ $x_1 \ldots x_n$ y) succeeds if y unifies with the sum $x_1 \ldots x_n$.
`-`	(- x y z) succeeds if x unifies with the difference of x and y.
`//`	(// x y z) succeeds if z unifies with the quotient of x and y.
`<`	(< x y) succeeds if x is less than y.
`>`	(> x y) succeeds if x is greater than y.
`<=`	(<= x y) succeeds if x is less than or equal to y.
`>=`	(>= x y) succeeds if x is greater than or equal to y.
`Disjoint`	(Disjoint x y) succeeds if the lists x and y have no common elements.
`Done`	(Done x t) succeeds if the result returned from executing the LISP expression x unifies with the term t.
`Element`	(Element x l) succeeds if the object x is an element of list l.
`ElementsIn`	(ElementsIn b s) succeeds if s is the set of elements in bag b.
`Inter`	(Inter x y z) succeeds if z is the intersection of lists x and y.
`Intersect`	(Intersect x y) succeeds if lists x and y have a non-empty intersection.
`MAnd`	(MAnd p l) succeeds if predicate p is true of every element in list l.
`MAndCan`	(MAndCan p l s) succeeds if s is the union of the lists y that satisfy (p x y) for each elememt in list l.
`MAndCar`	(MAndCar p l s) succeeds if s is the set of objects y that satisfy (p x y) for each elememt in list l.
`Member`	is a synonym for `Element`.
`MemList`	is a synonym for `Element`.
`SetDiff`	(SetDiff x y z) succeeds if z is set difference of x and y.

Subset	(Subset x y) succeeds if x is a subset of y.
Union	(Union x y z) succeeds if z is the list formed by appending x and y.

Another class of evaluable predicates is that of the metalinguistic predicates — predicates dealing with relations outside of the object-level universe that treat their arguments as syntactic objects without reference. Because of this property, one could only implement these predicates in 'pure' logic programming by using a vast table of all the tuples satisfying the predicate. Strictly speaking, the arithmetic predicates are also in this class. It includes the (self-explanatory) arithmetic predicates integer and number, and the numeric equality predicate num-=. This predicate works exactly like = except that when both its arguments are numeric it performs a comparison with a tolerance given by the value of num-=-threshold, which is initially 0.0001. Two other predicates allow examination of the binding status of variables and expressions:

Variable	(Variable <x>) succeeds if <x> is a currently unbound variable.
Ground	(Ground <p>) succeeds if the expression <p> contains no unbound variables.

A nice example of the use of Variable and Ground is the following (partial) implementation of an addition predicate that handles uninstantiated arguments:

```
(IF (AND (Ground $y)
         (Ground $z)
         (Variable $x)
         (- $z $y $x))
    (+ $x $y $z))
```

To facilitate interaction between MRS and LISP programs, a predicate (Value <x> <v>) is provided which succeeds when <v> is unifiable with the value of <x>. Also (Property <x> <v> <p>) succeeds when <x> has value <v> for property <p>. Asserting a Property or Value fact has the effect of setting the property or value.

§8.3 Predicate-related attachment — computable representations

The computable representation mechanism in MRS allows the user to specify classes of proposition for which the result of a lookup (and hence of a truep) is computed by a LISP subroutine, which will be known as the associated LISP subroutine (or ALS) for that class.

This is achieved by asserting a fact of the form

```
(repn <p> <rpn>)
```

which means that propositions matching <p> will be handled using the computable representation <rpn>. The assertion of the repn fact causes a tolookup and a tolookups fact to be stashed for the proposition class, which attaches it to the appropriate interface routines which call the ALS and then package the results into the proper binding list format expected by the theorem prover.

The ALS must be on the LISP property of the predicate involved, so one must assert the fact

```
(Property <pred> <ALS> LISP).
```

The <rpn> value used will be a mnemonic code specifying how the arguments are to be handled, whether a value is to be returned, whether binding lists are to be single or multiple and so on. The code will determine the interface routines to which the predicate will be attached. The rest of this section deals with the codes that MRS provides.

The basic thing to decide is whether the LISP routine is to return a value that is a function of its arguments, or to act as a predicate in itself (e.g. FLOATP). All function codes begin with F, all predicate codes begin with R (for Relation). if you always want a given predicate to be procedurally attached regardless of its arguments you can replace the repn and DEFPROP entries above by asserting

```
(relnproc <pred> <ALS> <rpn>)
```

for relations or

```
(funproc <pred> <ALS> <rpn>)
```

for functions.

Suppose we decide that the LISP routine is to return a value. The <rpn> code will begin with F. MRS predicate will need to have an extra argument which will be unified with the result when the function returns. Thus a LISP function (f $x_1 \ldots x_n$) will be attached to a predicate (f $x_1 \ldots x_n$ y), and y will be unified with the result returned by the function f. Often, we will want to use arguments which are themselves functional expressions. MRS allows for this in the second letter of the code which is E or Q. Q (for Quote) means that the arguments are passed 'as is' to the subroutine. E (for Eval) means that arguments which are terms beginning with predicates that have functional attachments are treated like functional expressions and evaluated by calling lookup on them with an extra argument. An example will make this clearer. Suppose we have a predicate (Average $a $b $avg) which we want to attach to a LISP function (Average a b). If we declare the predicate as having representation FE, then we can use it, or any other FE or FQ predicate, functionally as its first or second argument:

```
(Average (Average 2 4) 5 $x)
```

would succeed with $x bound to 4. As one might expect, we can also handle the case where we want to perform a num-= unification on the result rather than a straightforward =. To do this, use a third code letter A (for Arithmetic). We would probably want this for the averaging predicate so we would declare it by saying

```
(ASSERT '(funproc Average Average FEA)).
```

Now for the computable relations, which are a little more complicated. The first letter of the code will be R; the ALS will have the same number of arguments as the predicate The second letter is still E or Q as above. Plain RE and RQ relations are treated just as you would expect: if the LISP routine returns NIL the lookup (or truep) fails; if it returns a non-NIL value the lookup returns ((T . T)). For example, to attach FLOATP we would probably use the RE representation:

```
>(ASSERT '(relnproc FLOATP FLOATP RE))
DONE
>(TRUEP '(FLOATP 1.34))
NIL
```

Often we will want to be able to handle procedural attachments for predicates whose arguments may be unbound variables — the ALS will produce the correct bindings for the variables and return the appropriate binding list. If we add a B (for Binding) as the third letter of the code, this means that the ALS will return its own binding list (or NIL). Using this, we could write an averaging predicate that worked when say the first argument was unbound. To return binding lists, the ALS (and not MRS) will need to test its arguments to see if they are variables, perform the appropriate

computation accordingly and construct a binding list to return. This is made easier by the availablity of the system functions for handling variables and binding lists, which are described in chapter 10.

```
>(ASSERT '(relnproc Average AverageP REB))
DONE
>(DEFUN AverageP (x y avg)
   (COND ((AND (NUMBERP x) (NUMBERP y))
          (batchp (QUOTIENT (TIMES x y) 2) avg))
         ((AND (NUMBERP x) (blvarp y) (NUMBERP avg))
          (batchp (DIFFERENCE (PLUS avg avg) x) y))
         ((AND (blvarp x) (NUMBERP y) (NUMBERP avg))
          (batchp (DIFFERENCE (PLUS avg avg) y) x))))
AVERAGEP
>(TRUEP '(Average $x 3 4))
(($X . 5) (T . T))
```

In some cases, REB and RQB relations will want to return multiple solutions, for instance if we had a quadratic solver (Quad $a $b $c $x) which might want to return 0, 1 or 2 different bindings for $x. To do this, add a fourth letter M (for Multiple).

There exists one extra type of attachment which is a hybrid of the functional and relational classes. The RQFM representation allows for ALS routines which operate like functions on the first $n-1$ arguments but return a simple list of values which are to be interpreted as alternatives. The RQFM designation will cause a multiple binding list to be automatically constructed for the last argument (if unbound); if bound, the predicate will succeed if the last argument unifies with any of the returned values.

Thus, in summary, the available ways of getting LISP programs to do the work of predicate are FE, FQ, FEA for functions returning values, RQFM for multifunctions, and RE, RQ, REB, RQB, REBM, RQBM for relations.

To create one's own computable representation (if one can think of any other possibilities, that is) one should assert a fact of the form

```
(repn-method <rpn> <op> <method>)
```

for the <op>s lookup and lookups, where the <method> is an appropriate interface routine.

Chapter 9

Alternative representations and inference procedures

For certain computations and certain classes of facts, a representation other than the standard MRS indexed list structure is useful, and several such representations are provided. Moreover, some problems are best solved using inference procedures other than backward chaining with *modus ponens*. In some cases these procedures operate with the standard representation, in others they use specialized forms.

§9.1 Representations

The representations discussed here are essentially storage and retrieval methods for facts, and affect how the stored facts appear *internally*. If the database routines are well-written, the chosen representation need have no effect on the external appearance of the facts at all. A specific representation can be viewed as an implementation of an abstract data type with the operations `stash` and `lookup`. Different representations are more or less efficient for these operations for different classes of facts. MRS allows the user to specify that all facts matching a pattern <p> should be stored using representation <rpn> by asserting a fact of the form

```
(repn <p> <rpn>).
```

The effect of this is to cause the appropriate `tolookup` and `tostash` facts to be entered into the database. Since these facts operate at the meta-level the pattern <p> must use meta-level variables. The currently implemented representations are:

`pr`	The default; each fact is stored verbatim on the `pattern` property of a unique proposition (e.g. `P123`). The facts are then fully indexed on every position in the list structure of the fact. Only `pr` facts are accessible to the full range of theory-related commands (see chapter 7).
`cnf`	Conjunctive normal form; in this representations all facts are stored as disjunctions of literals (a literal is an atomic proposition or the negation of one). The whole database is an implicit conjunction of these disjunctions, hence the name. For example, `(IF A B)` is stored as the disjunction of `(NOT A)` and `B`. This representation is used by the resolution routines.
`dnf`	Disjunctive normal form; like CNF, but the database is a disjunction of conjunctions of literals. One can write a DNF resolution routine if desired.
`pl`	Property list representation; useful for storing the values of unary functions on (or attributes of) concepts; for example, `(Arity Member 2)` is stored by putting `2` on the `Arity` property of `Member`.
`tl`	is useful for storing unary relations such as `(IsWonderful MRS)`.
`dl`	is useful for storing many-many binary relations such as `SameAge`.

For example, suppose we wanted to use a lot of facts about people's ages. Since age is a many-one relation, we would use the `pl` representation for efficiency. To achieve this, enter

```
(ASSERT '(repn (Age &x &y) pl)).
```

Then, when we say

```
(STASH '(Age Nancy 91))
```

the attribute is stored directly on Nancy's property list:

```
(PLIST 'Nancy)
  (Age 91)
```

The `tl` representation also uses the property list directly, by putting a `T` on the appropriate property of the concept involved. The `dl` representation uses properties with multiple values stored as a list; for example, the property list of Henry VIII might end up as

```
(HasWife (KatharineOfA AnneB JeneS AnneOfC CatherineH KatherineP)).
```

With each of these property list representations the queries can have only the value uninstantiated, and variables in the facts will not be handled properly. This is an example of the generality/efficiency trade-off common in representations (and procedures for that matter).

The user can create her own representations by specifying the storage and retrieval routines for it; to do this one asserts facts of the form

```
(repn-method <rpn> <operation> <routine>)
```

for the operations `lookup`, `lookups` and `stash`. As an example, we'll take the case of storing attributes of objects identified by number, for instance a database of customer attributes using facts like

```
(CustomerName 3423 (John Q Public)).
```

To have these facts stored in an array, with its instant-access advantages, we would say

```
(ASSERT '(repn-method ar lookup ar-lookup))
(ASSERT '(repn-method ar lookups ar-lookups))
(ASSERT '(repn-method ar stash ar-stash))
(defun ar-lookup (p)
  (batchp (caddr p) (funcall (car p) (cadr p))))
(defun ar-lookups (p)
  ((lambda (bl) (cond (bl (list bl)) (t nil))) (ar-lookup p)))
(defun ar-stash (p)
  (apply* 'store (list (car p) (cadr p)) (caddr p)))
(ASSERT '(repn (CustomerName &number &name) ar))
```

Notice that the above routines only work if the customer number is already known. If this were not the case, they would have to be extended to handle queries such as `(CustomerName $n (A N Other))` by recognizing that the first argument to the predicate was a variable and then searching the entire array to find the index number for the given customer; for this class of query the array representation would be extremely inefficient. One solution would be to have `ar-assert` store the fact in both `ar` and `pr` representations, and have `ar-lookup` choose which to use according to the binding status of the arguments in `p`.

§9.2 Alternative inference procedures

MRS provides several modes of inference other than the standard backward chaining. Each is appropriate for a fairly ill-defined range of situations and deciding which to use is still something of an art. As with almost anything else, the user can write her own inference routines; the simplest to implement will be of the 'black-box' type, which take a proposition as argument and return a binding

list. They will be invoked using a totruep attachment and their internal workings will avoid the use of the scheduler architecture. All of the MRS-provided routines are of the non-black-box kind, putting their individual inference steps on the agenda so that the user can create control strategies to increase efficiency.

9.2.1 Forward chaining

Forward chaining has already been covered in an earlier chapter. There only remains to describe the task structure. The inference step task is very simple:

```
(fcdisp p)
```

where p is the proposition to be asserted and from which the forward chaining will take place. Multiple possibilities occur when a fact satisfies the premise of more than one rule, so that fcdisp tasks are placed on the agenda for each of the rule conclusions. A typical control strategy would give preference to the conclusion most likely to contribute to the desired goals.

9.2.2 Residue

The residue routine operates in a backward chaining fashion, the difference from the usual truep method being that it is allowed to assume the truth of propositions contained in assumable statements, or which can be proved to be assumable. Thus, if one stashes a fact (assumable <q>), then a call to (residue <p>) will do a backward-chaining proof of <p> and in doing so assume that <q> is true. A list of all the facts assumed in the proof of <p> is returned (residues <p>) returns all possible lists of such facts. Each list of facts is called the *residue* of <p>.

That's all very simple and straightforward I'm sure. What is not so clear to the uninitiated is what it's all for. Well, one use is for the expression of *default rules*. A typical example might be

```
(IF (AND (Bird $b) (UNPROVABLE (NOT (Flies $b)))
    (assumable (Flies $b))).
```

The use of residues and assumable facts has several advantages: default assumptions upon which a solution is based are distinguished from solid facts; the user is given a list of assumptions which he can check for validity in the individual cases; the addition of an exception to the database such as

```
(NOT (Flies OllyTheOstrich))
```

will not invalidate the rule, whereas if the conclusion were definite and not just assumable contradictions could arise, particularly if forward chaining or caching were in operation.

A second, and perhaps the major, use of residues is for synthesis of complex objects from specifications. For instance, we could take the rules for circuit behavior from chapter 5, tell the system it could assume any connections it wanted and call residue on the desired i/o pair and have it return the circuit it needed to assume to get the required behavior, described by a list of assumed facts looking very much like the descriptions of circuits as defined by the user. That, at least, is the idea.

Residues are implemented by the subroutine br. The task step is

```
(brdisp g1 a1 theory j1 ce)
```

where all the arguments are the same as for bcdisp except for theory, which contains the assumptions made so far in deriving the goal list g1.

9.2.3 Resolution

Unlike *modus ponens*, the resolution inference rule

$$\frac{(OR\ A_1 \ldots A_m),\ (OR\ B_1 \ldots B_n)}{(OR\ A_1 \ldots A_{i-1}\ A_{i+1} \ldots A_m\ B_1 \ldots B_{j-1}\ B_{j+1} \ldots B_n)\sigma} \quad \text{where } A_i\sigma = (NOT\ B_j\sigma) \text{ or } \textit{vice versa}$$

is complete, i.e. all possible logical conclusions of a set of facts can be drawn using the rule. This is probably the major reason why one would use it. From the definition two drawbacks are immediately apparent. Firstly, the rule itself is rather cumbersome and unintuitive in the sense that that normal human modes of reasoning do not fit it well. Secondly, it requires that the database be in CNF, which often renders facts unintelligible and/or greatly increased in size. However, for such applications as mathematical theorem-proving, resolution is still the method of choice, since with resolution one does not need to worry about the incompleteness or directionality of repesentation inherent in a rule-based system. With MRS's ability to convert facts to CNF automatically, resolution is a serious candidate for many applications. To prove a proposition using this method, the user should simply enter

```
(resolution <p>)
```

and the binding list will be returned just as with `truep`. `resolutions` needs no further comment. What happens is that <p> is first negated, then added to the database in CNF. Resolutions are then performed until an empty disjunction is created, signifying a contradiction. The rest of the database can be created in CNF by starting off with

```
(ASSERT '(repn &p cnf))
```

which causes all subsequently entered facts to be stored in normal form, and all retrievals to be performed accordingly.

The default strategy used is a set-of-support, linear-input strategy. The set-of-support strategy is the resolution equivalent of backward-chaining: only resolutions involving a goal clause or a clause whose derivation includes a goal clause are considered.

The standard resolution routine is `rs`, and the single inference step is

```
(rsdisp gl al theory)
```

where

`gl`	is a list of the clauses produced so far by resolution with the original negated-goal clauses
`al`	is the binding list for the variables in `gl`;
`theory`	is a locally-bound theory containing the original goal clauses.

9.2.4 Resolution with residues

`resolutionresidue` and `resolutionresidues` operate exactly as one would expect, performing resolution with assumptions returned. `rr` is the standard routine, the inference step is

```
(rrdisp gl cl al theory)
```

where the arguments are the same as for resolution with the addition of `cl` which is the list of assumptions made in deriving `gl`.

Chapter 10
Useful system functions

There are several places where the user will want to write some additional code of her own to tailor the system to her needs. These include writing procedural attachments for predicates, new computable representation interfaces, new representation storage and retrieval routines, formatting and filtering routines for returned solutions and, last but not least, new inference routines. Since MRS itself does all of these things, it is not surprising that it contains a lot of useful subroutines which are also available to the user.

§10.1 Testing for variables

In attached LISP subroutines, inference routines and often in ordinary MRS predicates one will want to test expressions to see if they are ground or variable.

`blvarp`	`(blvarp x)` returns `T` if `x` is a base-level variable, i.e. an atom beginning with `$`.
`mlvarp`	`(mlvarp x)` returns `T` if `x` is a meta-level variable, i.e. an atom beginning with `&`.
`varp`	`(varp x)` returns `T` if `x` any kind of variable.
`groundp`	`(groundp x)` returns `T` if `x` is an expression containing no variables.

§10.2 Matching and unification

These routines will come in useful in roughly the same areas as the variable testing routines. The one that is needed will depend on the level (base- or meta-) and the (non)necessity for standardization of variables.

`batchp`	`(batchp x y)` is a base-level unification routine that standardizes variables first, and returns the binding list (if any) for the variables in `x`.
`matchp`	`(matchp x y)` is a meta-level unification routine that standardizes variables first, and returns the binding list (if any) for the variables in `x`.
`samep`	`(samep x y)` is a base-level routine that returns a binding list for the variables in `x` if the two expressions are the same up to variable renaming.
`unifyp`	`(unifyp x y)` is a base-level unification routine that returns the most general unifier of `x` and `y` (if any) but does *not* distinguish occurrences of the same variable in the two expressions.

§10.3 Binding lists

Along with multiple uses in inference routines and attached predicates, the functions for binding list manipulation are very useful for formatting the solutions returned by `truep`.

`getvar`	`(getvar x bl)` returns the binding for variable `x` from binding list `bl`.
`getbdg`	`(getbdg x p)` = `(getvar x (truep p))`
`getbdgs`	is like `getbdg` but calls `trueps` and returns a list of values.
`getval(s)`	`(getval(s) '(r` $x_1 \ldots x_n$`))` = `(getbdg(s) y '(r` $x_1 \ldots x_n$ `y))`

`lookupbdg(s)`	is like `getbdg(s)` but uses `lookup(s)` instead of `truep(s)`.
`lookupval(s)`	is like `getval(s)` but uses `lookup(s)` instead of `truep(s)`.
`plug`	`(plug x bl)` returns expression `x` with its variables fully instantiated from the binding list `bl`.
`pluralize`	`(pluralize bl)` turns a single binding list into a multiple one by putting parentheses round it.
`singularize`	`(singularize bl)` turns a multiple binding list into a single one by taking the CAR of it.

§10.4 Tasks and the agenda

The following routines enable the user to write her own inference routines using the scheduler architecture:

`kb`	`(kb to<T>` $x_1 \ldots x_n$`)` invokes the meta-level backward-chainer `trtruep` to find out how to perform task `<T>` for the given arguments, then performs the task and returns the results.
`tb`	`(tb <T>` $x_1 \ldots x_n$`)` places the task `<T>` on the agenda with the given arguments.
`scheduler`	`(scheduler)` starts the deliberation-action process operating according to the flags `executable`, `executed` and `preferred`.

§10.5 Miscellaneous routines

The following fu: ctions and commands operate only on facts in the pr representation.

`datum`	`(datum p)` returns the proposition symbol for `p`.
`pattern`	`(pattern d)` returns the proposition for symbol `d`. Useful for unstashing facts without typing them in verbatim.
`mrsdump`	`(mrsdump th f)` writes out the propositions in theory `th` onto file `f` in such a way that they can be reloaded using a `LOAD`.
`mrsload`	`(mrsload f)` loads naked propositions (i.e. without stash commands) from file `f` into the current theory.
`mrssave`	`(mrssave` $th_1 \ldots th_n$ `f)` saves the propositions from the given theories onto the file `f` so that they can be reloaded using `mrsload`.

Chapter 11
Tracing, caching and justifications

§11.1 Tracing

At present the tracing mechanism in MRS is the only form of debugging other than the LISP-provided utilities, and it is somewhat inadequate to say the least. Once the system has chosen a task to perform (usually from the agenda), it will be printed out on the terminal if it matches with the pattern <p> provided by the user with the command

(TRACETASK <p>).

Normally the pattern will be just &x and the resulting output will list each bcdisp (or any other disp) task with its arguments in a format slightly more readable than that provided by TRACE. (UNTRACETASK <p>) turns off tracing for tasks matching <p>. To switch tracing off altogether just type (UNTRACETASK).

§11.2 Caching

In performing a proof, the system produces not only the final result but also several intermediate facts which are usually discarded without a second thought. This is often shockingly wasteful — these results may have to be recreated for some later proof, or they may even be interesting in themselves. MRS provides a simple caching facility whereby the built-in inference procedures (and others if they so desire) can stash specified classes of intermediate and final results.

To get this to happen, just set the value of the variable cache to the name of the theory you would like the results stashed in. See chapter 7 for a description of theories. To use the current theory (the value of theory) set cache to T. When the time comes to cache a result <p>, MRS tries to find a caching routine <r> such that

(tocache <p> <r>)

is true. The default is (cachebystash <p>) which behaves as described above. The user is free to change this as she wishes by unstashing the default and adding her own restricted caching classes or new caching routines. Due to the large number of results produced by even simple proofs it is advisable to put them in a separate theory which can then be easily emptied. To turn caching off, just type (SETQ cache NIL).

§11.3 Justifications

One of the trumpeted advantages of expert systems is their ability to explain their own reasoning. They achieve this feat by simply saving, for each deduction made, the premises, conclusions and inference rule used. This does not of course happen automatically; or rather, it does provided the justify flag is non-NIL. It should be set to the name of the theory in which you would like the justifications stashed. If this sounds similar to caching, there's more to come. Not only does the justify flag cause saving of justifications, it also causes all the results that caching would cache to be saved on the property lists of specially-generated proposition symbols which aren't attached to any theory at all. This is because the justifications refer to these intermediate results, and they must be available for why and where (see below) to print them out.

A justification is a fact of the form

(just <concl> <method> <premise$_1$> ... <premise$_n$>)

where the conclusion and premises (which include the rule used) are represented by their corresponding proposition symbols, and the method is bc, fc or whatever.

The justifications are used by the two commands **why** and **where**. (**why** <p>), if a justification exists for the proposition, prints out the inference method and the premises from which the proposition was deduced. (**where** <p>) gives the same informations for all inferences in which <p>) took part as a premise. The following simple example illustrates these ideas:

```
>(SETQ justify T)    ;;Will use current theory for stashing justifications
T
>theory
GLOBAL
>(STASH '(Man Socrates))
P285
>(STASH '(IF (Man $x) (Mortal $x)))
P286
>(TRUEP '(Mortal Socrates))
((T . T))
>(why '(Mortal Socrates))
P288: (MORTAL SOCRATES) by BC
    P285: (MAN SOCRATES)
    P286: (IF (MAN $X) (MORTAL $X))
DONE
>(where '(Man Socrates))
P288: (MORTAL SOCRATES) by BC
    P285: (MAN SOCRATES)
    P286: (IF (MAN $X) (MORTAL $X))
DONE
>(PRFACTS 'just)
P287: (JUST P285 BC)
P289: (JUST P288 BC P285 P286)
DONE
```

Chapter 12

More general input and output

The view provided so far of the MRS system is that of a 'naked' theorem prover, taking facts and queries in predicate calculus as input and printing facts and binding lists as output. However, since all the output mechanisms are implemented using default rules, the user can build any desired interface for displaying facts. In addition, MRS provides a mechanism for asking intelligible questions of the user; thus it is quite easy to build a system in which the user never has to see any predicate calculus at all. Also described in this chapter are methods for monitoring the progress of inference on a display, and for directly editing facts in the database.

§12.1 Asking questions of the user

Logic programming is particularly suited to the implementation of *consultation systems* — systems that operate in the same mode as a human consultant, by being informed of the user's overall need and then asking appropriate questions to determine the necessary information for the solution of the problem.

The backward chaining approach to consultation makes the problem the initial goal, and works back through the rules until it finds premises that can be supplied by the user, rather like having an extra database accessible via the terminal instead of the `lookup` routine. We can use the procedural attachment mechanism of MRS to implement this idea by having a special subroutine `ask(s)` perform the function of `truep(s)` for the class of facts that the user is likely to know; for example an interactive tictactoe program might have

```
(totruep (YourMove &x) ask).
```

First of all `ask` calls `output` on its argument <p> to display it in an understandable form (see next section) and prints it out. Then it examines <p> to see if it can be answered by a yes/no, which is the case if it has no free variables. If so, it asks the user if the proposition is true and returns `((T . T))` or `NIL` accordingly. If not, it asks the user to supply a value for each variable so as to make the proposition true. The routine `asks`, which simulates `trueps`, prompts the user for all the sets of values of the variables which satisfy the query. In most cases, unless `truep` is called directly by the user on an askable query (which would be somewhat self-defeating), the routine used will be `asks` rather than `ask`. However, many relations are really functional (i.e. have only one satisfying binding) so the questions asking for multiple values can be a little irritating. For instance, one is hardly likely to want to make more than one move at a time in a game of tictactoe. The routines can accommodate this information if the user states that the predicate involved is actually a function:

```
(STASH '(Function YourMove)).
```

This will cause a single value only to be prompted for.

Often a proposition will be used as a premise of more than one rule. If it is an askable one, this may have the annoying effect of causing the user to be asked the same question twice (if not more). One solution is to tell MRS that the class of askable propositions is also to be cached - `ask` won't ask something if it is already in the database. Furthermore, this can be used to produce a theory containing all of the facts about a particular user that pertain to her problems which can then easily be stored permanently using `mrsave`.

§12.2 Displaying facts

All of the MRS routines that print facts call `output` or `outputs` to do so. The argument to `output` is a single fact, that to `outputs` a list of facts. Facts are printed out by `facts`, `contents`, the tracing

These facilities are best illustrated by running the output demo, which may be invoked by loading the file OUTPUT-T. A number of output routines are implemented on the Symbolics machine only, since they take advantage of its graphics capabilities. These are illustrated in the output demonstration for the Symbolics machine.

§12.3 Monitoring

Particularly in a system using condition-action rules, or one using forward chaining, one will want to monitor the changes made to the database by assertions. For instance, a flight simulator will want to continuously monitor such facts as the amount of fuel remaining and the current altitude. MRS provides a mechanism for doing this using demons (see chapter 6). Once a proposition is being monitored, any assertion matching the proposition will cause a function to be invoked that can either output the assertion directly or call a specialised routine for updating the appropriate display.

Monitoring can be initiated for a class of facts matching <p> using the command (monitor <p>). The command monitors takes a list of propositions to be monitored. The effects are as follows. Firstly, forward chaining is asserted for the proposition so that the monitor demon will fire:

```
(toassert <p> fc).
```

Secondly, the demon is created:

```
(IF <p> (runnable (monitor-hook <p>))).
```

Now we have to decide if we're going to monitor the fact by just printing it out, or by maintaining some display (such as a needle gauge or a digital readout) for it. To discover this, trtruep is called on a goal of the form

```
(tomonitor <p> <method>).
```

It is important to remember that the call to monitor is intended to *initiate* monitoring for its argument, rather than actually perform it. Thus the tomonitor method, which will of course be a LISP routine, set up whatever display window or file may be needed for monitoring the facts. monitor-hook is the function that gets the facts displayed, and the tomonitor method must also therefore provide some information for monitor-hook as to exactly how and where to display them. The convention adopted is that it should return a CONSed pair whose CAR is the name of the function which will do the displaying, and whose CDR indicates where these facts are to be displayed (it might be the name of a file, or a pointer to a window). The monitor function takes this returned pair and uses it to create a fact in the database of the form

```
(mdisplay <p> <display function> <display argument>).
```

Now, whenever a fact matching <p> is asserted, monitor-hook will find the mdisplay fact for it and call the display function with two arguments: the fact and the display argument. Multiple displays for a fact can be handled because monitor-hook finds *all* the applicable mdisplay facts.

Clearly, the desired default display function must be output. Since output doesn't require the display argument, our default tomonitor method will just be a function that returns (output . NIL), i.e.

```
(tomonitor &p (lambda (p) '(output)))
```

facility, the justification routines why and where, and by ask. Moreover, the user can invoke output directly for display and debugging purposes.

The output(s) routines work by calling trtruep (the meta-level theorem prover) on the goal

```
(tooutput(s) <p> <m>)
```

where <p> is the fact(s) to be printed. The returned output method <m> will be called on the fact(s); thus the user can specify any desired output routine by stashing a tooutput(s) fact for it. The output routines can do anything the user wishes, from printing charts and trees to moving dials or beeping Morse code. Several useful routines are provided, and described below.

The default tooutput method is pnl-output (pnl standing for pseudo-natural language). The default tooutputs method is prop-outputs, which prints the facts verbatim, preceded by their associated proposition symbol (as in the output from the facts routine). Sometimes, for instance during debugging, one may want to print out facts in this default format instead of whatever fancy format one has defined for them. The command (prfacts <t> <level>) (the arguments are the same as those of facts) does just this.

12.2.1 Pseudo-natural language output

The pnl-output routine mentioned above is designed to provide a form of translation from predicate calculus into English (or whatever) using *templates*. A template is essentially a sentence with holes that are to be plugged with appropriate values. For instance, we would specify the template for YourMove by stashing

```
(Template (YourMove &x) (Your next move is going to be &x)).
```

When output is called on a proposition p, it tries to find a template whose left-hand side unifies with p (note that this will be a meta-level unification) and when successful applies the binding list to the right-hand side and returns the result. This process is actually recursive; pnl-output calls itself on each binding of a template variable before plugging it into the sentence. Thus suppose we had a case in the tictactoe game where the program could predict where the user was going to move and wanted to show off. Recalling that the moves are represented by the digits 1 to 9, we could add

```
(Template 1 (in the top left corner))
...
(Template 5 (in the center))
...
(Template 9 (in the bottom right corner)).
```

12.2.2 Other output routines

Several output routines are provided with MRS for displaying information in various formats. Each of the following functions takes a list of facts as its only argument:

indent-tree-outputs Prints out the binary relation described by the facts in the list as an indented tree, if possible.

simple-bar-outputs Takes a list of facts describing a function with numeric range, and prints out a simple bar chart containing the information.

table-outputs Sorts the list of facts by predicate and arity, and prints out a table for each relation.

This has all been very confusing, no doubt, but perhaps an example will help. The following transcript comes from the output facility demo, and shows how to implement a history file of database assertions using monitoring (while this isn't the most exciting use of monitoring, it is, at least, implementable in all versions of MRS):

```
MONITORING for dumb terminals - count up to ten->>

(STASH (QUOTE (IF (AND (NEAR-TEN $X) (< $X 10) (+ $X 1 $Y)) (NEAR-TEN $Y))))
(TRSTASH (QUOTE (TEMPLATE (NEAR-TEN &X) (&X  is near ten))))

(MONITOR (QUOTE (NEAR-TEN $X)))
 $X  is near ten

(TRSTASH (QUOTE (TONONITOR (NEAR-TEN &X) HISTORY-MONITOR)))
(MONITOR (QUOTE (NEAR-TEN $X)))

(DEFUN HISTORY-MONITOR (P &OPTIONAL (FILE NIL))
  (COND ((NULL FILE)
         (LET ((STREAM (OPEN (QUOTE OUTPUT-DEMO) (QUOTE OUT))))
              (HISTORY-MONITOR P STREAM)
              (CONS (QUOTE HISTORY-MONITOR) STREAM)))
        ((EQ P (QUOTE KILL)) (CLOSE FILE))
        (T (PRINC P FILE) (TERPRI FILE))))

->>
(ASSERT (QUOTE (NEAR-TEN 3)))
 3  is near ten
 4  is near ten
 5  is near ten
 6  is near ten
 7  is near ten
 8  is near ten
 9  is near ten
 10  is near ten

(UNMONITOR (QUOTE (NEAR-TEN $X)))

(LET ((IN (OPEN (QUOTE OUTPUT-DEMO) (QUOTE IN))))
     (PRINC (READ IN)) (CLOSE IN))
(NEAR-TEN $X)
```

First we stash a simple rule for counting up to ten, and a template for printing its status. Then we call monitor to establish default monitoring for NEAR-TEN, which will display it using output. Then we add an additional monitor using the function history-monitor, which will print the facts directly on a file. Using an optional argument, history-monitor can act as both the set-up and the display function. Forward chaining for NEAR-TEN is turned on by monitor, so when we assert (NEAR-TEN 3) it counts up to ten, and each new fact is both displayed on the terminal and printed on the file. At the end, we call unmonitor on the fact to terminate monitoring, and check the output file to see that something was in fact printed on it.

§12.4 Editing

At present MRS provides for direct database editing only on the Symbolics Lisp Machine, although it is anticipated that this will soon be extended to the other implementations. There are two editing commands:

`facts-edit`	`(facts-edit <t> <n>)` finds all facts containing the term `<t>` (up to the optional level `<n>`), and calls the appropriate `toedit` method for those facts.
`contents-edit`	`(contents-edit <th>)` finds and edits all facts in the theory `<th>` (which defaults to the value of `theory` if omitted).

It is the job of the editing function to invoke the editing environment, and to return a list of all the facts in their new form, including those remaining unchanged; if none are changed, the editing function may return `NIL`.

It is also possible to do editing by using an output routine which retains control, allows the user to modify the facts displayed, and updates the database before returning. Thus the user can enter information by merely mouse-setting a dial or gauge, as well as typing facts explicitly in an editor.

Appendix A :

Answers to exercises

§A.1 Answers to problems from chapter 2

1. `(Horse Dobbin)`
   Dobbin is a horse.
2. `(Member Dobbin Horses)`
   Dobbin is a member of the class of horses.
3. `(NOT (Horse Dobbin))`
   Dobbin is not a horse.
4. `(OR (Horse Dobbin) (Donkey Dobbin))`
   Dobbin is a horse or a donkey.
5. `(IF (Horse Dobbin) (Mammal Dobbin))`
   If Dobbin is a horse then he is a mammal.
6. `(IF (Horse $x) (Mammal $x))`
   All horses are mammals.
7. `(IF (OR (Horse $x) (Cow $x)) (FourLegged $x))`
   All horses and cows are four-legged.
8. `(AND (Mammal $x) (FourLegged $x))`
   Everything is a four-legged mammal.
9. `(IF (NOT (Horse Dobbin)) (Dutchman Ermintrude))`
   If Dobbin is not a horse then Ermintrude is a Dutchman.
10. `(IF (NOT (Cow $x)) (Brown Dobbin))`
    If there is anything that isn't a cow then Dobbin is brown.
11. `(IF (AND (Horse $x) (NOT (Mammal $x))) (Cow $x))`
    Every horse that isn't a mammal is a cow.
12. `(IF (OR (= $x Dobbin) (= $x Tonto)) (Horse $x))`
    Dobbin and Tonto are horses.
13. `(IF (AND (= $x Dobbin) (= $x Tonto)) (Horse $x))`
    Everything which is both Dobbin and Tonto is a horse.
14. `(IF (AND (Mammal $x) (NOT (= $x Dobbin))) (NOT (Horse $x))`
    Dobbin is the only mammal that's a horse.
15. `(IF (AND (Horse $x) (NOT (= $x $y))) (NOT (Horse $y))`
    There is at most one horse. (Sometimes propositions are difficult to translate directly — it is better then to construct a universe that is consistent with them and describe that universe.)
16. `(NOT (Member $x $x))`
    Nothing is a member of itself.
17. `(IF (AND (Horse $x) (Brown $x)) (Brown (TailOf $x)))`
    Brown horses have brown tails.

18. All dogs bark at their neighbours' dogs.

```
(IF (AND (Dog $d) (Neighbour $d $n) (BelongsTo $nd $n) (Dog $nd))
    (BarksAt $d $nd))
```

19. No real numbers are integers.

```
(IF (RealNumber $x) (NOT (Integer $x)))
```

20. Horses who hate dogs like ice cream.

```
(IF (AND (Horse $h) (Hates $h Dogs))
    (Likes $h IceCream))
```

21. Giraffes have longer necks than Dobbin.

```
(IF (Giraffe $g) (Longer (NeckOf $g) (NeckOf Dobbin)))
```

22. An-An is the only male panda in London.

```
(IF (AND (Male $x) (Panda $x) (In $x London))
    (= $x An-An))
```

23. Zero is an integer.

```
(Integer 0)
```

24. The fractional part of an integer is zero.

```
(IF (Integer $x) (FractionalPart $x 0))
```

25. The product of two real numbers is a real number.

```
(IF (AND (RealNumber $x) (RealNumber $y) (* $x $y $z))
    (RealNumber $z))
```

26. The product of a positive integer and its inverse is unity.

```
(IF (AND (Integer $x) (> $x 0) (Quotient 1 $x $inv))
    (* $x $inv 1))
```

27. Zero is an additive identity.

```
(+ $x 0 $x)
```

28. The product of two real numbers is never an imaginary number.

```
(IF (AND (RealNumber $x) (RealNumber $y) (* $x $y $z))
    (NOT (Imaginary $z)))
```

29. All numbers are either real or imaginary or both.

```
(IF (Number $x) (OR (RealNumber $x) (Imaginary $x)))
```

30. All Englishmen, Scotsmen and Welshmen are British.

```
(IF (AND (EnglishPerson $m) (ScotsPerson $m) (WelshPerson $m))
    (BritishPerson $m))
```

§A.2 Answers to problems from chapter 3

In the following answers we will rename variables in those cases where it is necessary as $1, $2 etc., corresponding to the variables in the second proposition in left to right order of appearance.

1. (p $a) and ($r x).
   (($r . p) ($a . x))
2. (p $a) and ($a q).
   (($a . q) ($1 . p))
3. (p $a c) and (p $y).
   No unifier possible — different numbers of arguments.
4. (q (f $c)) and (q $d).
   (($d . (f $c)))
5. (r (g $c)) and (r $c).
   (($1 . (g $c)))
6. (r $x (h $x)) and (r $b $b).
   No possible unifier — infinite substitution process.
7. (p $a (g $a) (h $a)) and (p (g $b) (g . $c) ($d . $c)).
   (($a . (g $b)) ($c . ((g $b))) ($d . h))
8. (q $a) and ($r . $s).
   (($r . a) ($s . ($a)))
9. (r $b . $b) and (r $c a).
   (($b . (a)) ($c . (a)))

§A.3 Answers to problems from chapter 5

1. Tictactoe solution.

```
;Strategy rules - last has precedence because of reverse search order

(IF (AND (Square $move) (unknown (On $anyside $move)))
    (BestMove $OX $move))
(IF (AND (OppositeSide $OX $opponent) (ImmediateWin $opponent $move))
    (BestMove $OX $move))
(IF (ImmediateWin $OX $move) (BestMove $OX $move))

;Rules for deciding when an immediate win is available

(IF (AND (WinLine $A $B $C)
         (On $OX $A)
         (On $OX $B)
         (unknown (On $X $C)))
    (ImmediateWin $OX $C))
(IF (AND (WinLine $A $B $C)
         (On $OX $A)
         (On $OX $C)
         (unknown (On $X $B)))
    (ImmediateWin $OX $B))
(IF (AND (WinLine $A $B $C)
         (On $OX $C)
         (On $OX $B)
         (unknown (On $X $A)))
    (ImmediateWin $OX $A))

(OppositeSide O X)
(OppositeSide X O)

;Squares are listed in this order so the system will
;pick the generally better squares when it has no other clue.

(Square 2)
(Square 4)
(Square 6)
(Square 8)
(Square 1)
(Square 3)
(Square 7)
(Square 9)
(Square 5)

(WinLine 1 2 3)
(WinLine 1 5 9)
(WinLine 1 4 7)
```

```
(WinLine 7 5 3)
(WinLine 4 5 6)
(WinLine 7 8 9)
(WinLine 2 5 8)
(WinLine 3 6 9)
```

2. Chess solution.

This problem is not conceptually difficult but is good practice for organizing a large, complex set of rules. It is important to think carefully about what predicates to define so that each has a clear meaning to you and also is a useful building block for the expression of the higher level rules.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The following rules define the LegalMove predicate which tests ;
; or generates legal moves in chess. The arguments of LegalMove  ;
; are as follows:                                                 ;
;                                                                 ;
;   $bw       - The color of the side whose turn it is to move.  ;
;               Values are Black and White.                       ;
;   $col,$row - The column and row coords of the origin square.  ;
;               White's QR1 is designated 1,1.                    ;
;   $newcol,$newrow - The coords of the destination square.      ;
;   $flag     - Indicates the move class:                         ;
;                   O is a normal move                            ;
;                   P is an en passant capture                    ;
;                   C is a castling move.                         ;
;                   N/B/R/Q is the new piece for a queening move  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Rules establishing the various classes of moves and their legality ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;Condition for a normal move to be legal (king moves done separately):
;   not already in check, and no discovered check

(IF (AND (Unknown (InCheck $bw))
         (Move $bw $col $row $newcol $newrow)
         (On $bw $piece $col $row)
         (Unknown (= $piece K))
         (Unprovable (DiscoveredCheck $bw $col $row $newcol $newrow)))
    (LegalMove $bw $col $row $newcol $newrow O))

;If castling is possible it's legal

(IF (CastlingMove $bw $side)
    (LegalMove $bw Castles $side  0 0 C))

;Condition for a king move to be safe

(IF (AND (On $bw K $col $row)
         (Move $bw $col $row $newcol $newrow)
         (Opponent $bw $wb)
         (Unprovable (Attacking $wb $newcol $newrow))
         (- $newcol $col $cv)
```

```
        (- $newrow $row $rv)
        (Unprovable
          (AND (AttacksAlong $wb $piece $pcol $prow $col $row $cv $rv)
               (MultiPiece $piece))))
    (LegalMove $bw $col $row $newcol $newrow 0))

;Condition for a move to get us out of check
;  (other than a king move, which is tested for safety already)
;  This can only work if there is only one checking piece, which is
;  on $pcol,$prow

(IF (AND (InCheck $bw)
         (Opponent $bw $wb)
         (On $bw K $kcol $krow)
         (SETOF ($pc $pr)
                (Attacks $wb $p $pc $pr $kcol $krow)
                (($pcol $prow)))
         (EscapesCheck $pcol $prow $bw $col $row $newcol $newrow $flag))
    (LegalMove $bw $col $row $newcol $newrow $flag))

;Condition for a pawn move to be legal

(IF (AND (Unknown (InCheck $bw))
         (PawnMove $bw $col $row $newcol $newrow $qpiece)
         (Unprovable (DiscoveredCheck $bw $col $row $newcol $newrow)))
    (LegalMove $bw $col $row $newcol $newrow $qpiece))

;Condition for an en passant move to be legal

(IF (AND (Unknown (InCheck $bw))
         (EnPassantMove $bw $col $row $newcol $newrow P)
         (Unprovable (DiscoveredCheck $bw $col $row $newcol $newrow))
         (Unprovable (Pinned $bw $newcol $row $pcol $prow)))
    (LegalMove $bw $col $row $newcol $newrow P))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Conditions for various types of moves to escape check, either by ;
;  taking the checking piece or by interposing                      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (AND (PawnMove $bw $col $row $col $newrow $qpiece)
         (On $bw K $kcol $krow)
         (Between $col $newrow $kcol $krow $pcol $prow)
         (Unprovable (DiscoveredCheck $bw $col $row $col $newrow)))
   (EscapesCheck $pcol $prow $bw $col $row $col $newrow $flag))

(IF (AND (PawnMove $bw $col $row $pcol $prow $qpiece)
         (Unprovable (DiscoveredCheck $bw $col $row $col $newrow)))
```

```
   (EscapesCheck $pcol $prow $bw $col $row $pcol $prow $qpiece))

(IF (AND (EnPassantMove $bw $col $prow $pcol $newrow P)
         (Unprovable (DiscoveredCheck $bw $col $prow $pcol $newrow))
         (Unprovable (Pinned $bw $pcol $prow $pinc $pinr)))
   (EscapesCheck $pcol $prow $bw $col $prow $pcol $newrow P))

(IF (AND (EnPassantMove $bw $col $row $newcol $newrow P)
         (On $bw K $kcol $krow)
         (Between $newcol $newrow $kcol $krow $pcol $prow)
         (Unprovable (DiscoveredCheck $bw $col $row $newcol $newrow)))
   (EscapesCheck $pcol $prow $bw $col $row $newcol $newrow P))

(IF (AND (Move $bw $col $row $newcol $newrow)
         (UNKNOWN (On $bw K $col $row))
         (Unprovable (DiscoveredCheck $bw $col $row $newcol $newrow))
         (On $bw K $kcol $krow)
         (OR (Between $newcol $newrow $kcol $krow $pcol $prow)
             (AND (= $newcol $pcol) (= $newrow $prow))))
   (EscapesCheck $pcol $prow $bw $col $row $newcol $newrow 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Conditions for legal castling: unmoved king and rook, not in check, ;
;  no intervening pieces, no intervening checks.                      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (AND (UnmovedK $bw)
         (CastlingSide $side)
         (Unknown (InCheck $bw))
         (UnmovedR $bw $side)
         (Opponent $bw $wb)
         (Unprovable (AND (CastlingMoveSquare $bw $side $col $row)
                          (On $anyside $anypiece $col $row)))
         (Unprovable (AND (CastlingCheckSquare $bw $side $col $row)
                          (Attacking $wb $col $row))))
    (CastlingMove $bw $side))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Conditions for pawn moves other than en passant ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (AND (On $bw P $col $row)
         (Direction $bw $oneforward)
         (Opponent $bw $wb)
         (PawnRank $wb $qrank)
         (OR (AND (= $col $newcol)
                  (+ $row $oneforward $nextrow)
                  (Unknown (On $anyside $anypiece $col $nextrow))
```

```
                (OR (= $nextrow $newrow)
                    (AND (PawnRank $bw $row)
                         (+ $nextrow $oneforward $newrow)
                         (Unknown
                           (On $anyside2 $anypiece2 $col $newrow)))))
            (AND (Attacks $bw P $col $row $newcol $newrow)
                 (On $wb $anypiece3 $newcol $newrow)))
        (OR (AND (= $row $qrank)
                 (QueeningPiece $qpiece))
            (AND (Unknown (= $row $qrank))
                 (= $qpiece 0))))
    (PawnMove $bw $col $row $newcol $newrow $qpiece))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Condition for a normal move other than pawn moves ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (AND (On $bw $piece $c1 $r1) (Unknown (= $piece P))
         (Attacks $bw $piece $c1 $r1 $c2 $r2)
         (Unknown (On $bw $anypiece  $c2 $r2)))
    (Move $bw $c1 $r1 $c2 $r2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Condition for en passant move ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (AND (EnPassantRank $bw $row)
         (On $bw $col $row)
         (PreviousMove $wb $pcol $prow $pcol $row 0)
         (PawnRank $wb $prow)
         (On $wb P $pcol $row)
         (Attacks $bw P $col $row $pcol $newrow))
    (EnPassantMove $bw $col $row $newcol $newrow P))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Rules for determining discovered checks ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(IF (AND (Pinned $bw $col $row $pcol $prow)
         (- $newcol $col $cv1)
         (- $newrow $row $rv1)
         (- $pcol $col $cv2)
         (- $prow $row $rv2)
         (Unprovable (Parallel $cv1 $rv1 $cv2 $rv2)))
    (DiscoveredCheck $bw $col $row $newcol $newrow))

;Condition for a piece to be 'pinned', i.e. unremovable by $bw
;[pcol,prow]>>>>>>>>>>[col,row][nextcol,nextrow]>>>>>>>>>>[kcol,krow]
```

```
(IF (AND (On $bw K $kcol $krow)
         (UnitVector $col $row $kcol $krow $cv $rv)
         (+ $col $cv $nextcol) (+ $row $rv $nextrow)
         (Route $nextcol $nextrow $kcol $krow $cv $rv)
         (Opponent $bw $wb)
         (AttacksAlong $wb $piece $pcol $prow $col $row $cv $rv)
         (MultiPiece $piece))
    (Pinned $bw $col $row $pcol $prow))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Rules to determine when pieces are attacking squares ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;Definitions of different specificities of attacks

(IF (AttacksAlong $bw $p $c1 $r1 $c2 $r2 $cv $rv)
    (Attacks $bw $p $c1 $r1 $c2 $r2))

(IF (Attacks $bw $p $c1 $r1 $c2 $r2)
    (Attacking $bw $c2 $r2))

;Condition for attacking a 'neighbouring' square with vector $cv $rv

(IF (AND (On $bw $piece $col $row)
         (MoveVector $piece $bw $cv $rv)
         (NextTo $col $row $cv $rv $newcol $newrow))
    (AttacksDirectly $bw $piece $col $row $newcol $newrow $cv $rv))

;Base case for attacking a 'distant' square

(IF (AND (AttacksDirectly $bw $piece $col $row $newcol $newrow $cv $rv)
         (Unknown (MultiPiece $piece)))
    (AttacksAlong $bw $piece $col $row $newcol $newrow $cv $rv))

;Condition for a piece to attack a 'distant' square with vector $cv $rv

(IF (AND (On $bw $piece $col $row)
         (MultiPiece $piece)
         (AttacksDirectly $bw $piece $col $row $col2 $row2 $cv $rv)
         (Route $col2 $row2 $newcol $newrow $cv $rv))
    (AttacksAlong $bw $piece $col $row $newcol $newrow $cv $rv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Route sees if there is a clear path from one ;
;     square to another along a given vector.  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(Route $col $row $col $row $cv $rv))
```

```
(IF (AND (Unknown (On $anyside $anypiece $col $row))
         (Route $col2 $row2 $newcol $newrow $cv $rv))
    (Route $col $row $newcol $newrow $cv $rv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Predicates for handling vector arithmetic ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; NextTo gives the next square to $col $row along vector $cv $rv

(IF (AND
         (+ $col $cv $newcol)
         (< $newcol 9) (> $newcol 0)
         (+ $row $rv $newrow)
         (< $newrow 9) (> $newrow 0))
    (NextTo $col $row $cv $rv $newcol $newrow))

;Definition of parallel vectors

(IF (IS (- (* $cv1 $rv2) (* $cv2 $rv1)) 0)
    (Parallel $cv1 $rv1 $cv2 $rv2))

;UnitVector finds the appropriate move vector to get between two squares

(IF (AND (- $col2 $col1 $mcv)
         (- $row2 $row1 $mrv)
         (Sign $mcv $cv)
         (Sign $mrv $rv))
    (UnitVector $col1 $row1 $col2 $row2 $cv $rv))

(IF (> $x 0) (Sign $x 1))
(IF (< $x 0) (Sign $x -1))
(IF (= $x 0) (Sign $x 0))

;Between sees if a square x is between two others y,z

(IF (AND (- $zc $xc $cv1)
         (- $zr $xr $rv1)
         (- $yc $xc $cv2)
         (- $yr $xr $rv2)
         (Parallel $cv1 $rv1 $cv2 $rv2)
         (IS (+ (* $cv1 $cv2) (* $rv1 $rv2)) $dotprod)
         (< $dotprod 0))
    (Between $xc $xr $yc $yr $zc $zr))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Data tables for castling, move vectors, piece classes ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(CastlingSide Kings-side)
(CastlingSide Queens-side)
(Opponent White Black)
(Opponent Black White)
(CastlingCheckSquare White Kings-side 7 1)
(CastlingCheckSquare White Kings-side 6 1)
(CastlingCheckSquare White Queens-side 3 1)
(CastlingCheckSquare White Queens-side 4 1)
(CastlingCheckSquare Black Kings-side 7 8)
(CastlingCheckSquare Black Kings-side 6 8)
(CastlingCheckSquare Black Queens-side 3 8)
(CastlingCheckSquare Black Queens-side 4 8)
(CastlingMoveSquare White Kings-side 7 1)
(CastlingMoveSquare White Kings-side 6 1)
(CastlingMoveSquare White Queens-side 2 1)
(CastlingMoveSquare White Queens-side 3 1)
(CastlingMoveSquare White Queens-side 4 1)
(CastlingMoveSquare Black Kings-side 7 8)
(CastlingMoveSquare Black Kings-side 6 8)
(CastlingMoveSquare Black Queens-side 2 8)
(CastlingMoveSquare Black Queens-side 3 8)
(CastlingMoveSquare Black Queens-side 4 8)
(Direction White 1)
(Direction Black -1)
(PawnRank White 2)
(PawnRank Black 7)
(EnPassantRank White 5)
(EnPassantRank Black 4)
(QueeningPiece N)
(QueeningPiece B)
(QueeningPiece R)
(QueeningPiece Q)
(MultiPiece B)
(MultiPiece R)
(MultiPiece Q)
(MoveVector P White 1 1)
(MoveVector P White -1 1)
(MoveVector P Black 1 -1)
(MoveVector P Black -1 -1)
(MoveVector N $anyside 1 2)
(MoveVector N $anyside 2 1)
(MoveVector N $anyside 2 -1)
(MoveVector N $anyside 1 -2)
(MoveVector N $anyside -1 -2)
(MoveVector N $anyside -2 -1)
(MoveVector N $anyside -2 1)
(MoveVector N $anyside -1 2)
```

```
(MoveVector B $anyside 1 1)
(MoveVector B $anyside 1 -1)
(MoveVector B $anyside -1 -1)
(MoveVector B $anyside -1 1)
(MoveVector R $anyside 1 0)
(MoveVector R $anyside 0 1)
(MoveVector R $anyside -1 0)
(MoveVector R $anyside 0 -1)
(MoveVector Q $anyside 1 1)
(MoveVector Q $anyside 1 -1)
(MoveVector Q $anyside -1 -1)
(MoveVector Q $anyside -1 1)
(MoveVector Q $anyside 1 0)
(MoveVector Q $anyside 0 1)
(MoveVector Q $anyside -1 0)
(MoveVector Q $anyside 0 -1)
(MoveVector K $anyside 1 1)
(MoveVector K $anyside 1 -1)
(MoveVector K $anyside -1 -1)
(MoveVector K $anyside -1 1)
(MoveVector K $anyside 1 0)
(MoveVector K $anyside 0 1)
(MoveVector K $anyside -1 0)
(MoveVector K $anyside 0 -1)
```

3. Geometry solution.

The ontology for geometry is fairly well-known — points, lines, angles and circles cover most things. Points are identified by constant symbols just as in real life. Lines and angles can be represented by terms with function symbols `Line` and `Angle`; for example, the line segment $\overline{AB}$ wil be called `(Line A B)`. Circles could be represented by a term with function symbol `Circle` with the points on the circle as arguments, but for our purposes, as is common in geometry, this won't be needed. It is important to note that the angle terms refer to the particular piece of angle subtended by the particular three points, so our geometrical theorems will not state that angles are equal, but that their sizes are equal; similarly, lines are not equal, but have equal length.

A real difficulty arises with the use of an angle described by a term such as `(Angle A B C)`: how is MRS to know that it is the same as `(Angle C B A)`? The same problem arises with `(Line A B)` and `(Line B A)`. Basically whenever we want to use a term such as `(Angle $a $b $c)` in a rule, we are obliged to write the same r le again but with the points in reverse order, in case the facts we have about that angle happen to be expressed that way. It would be nice if we could get the unification routine to treat these as unifiable, then we could write rules and describe problem instances as if there were no problem at all, but that is, as they say, beyond the scope of this book. There is, however, a way of achieving the same effect. What we want to have is a term that will unify with a given angle whichever way round it is written; to achieve this we use a constructor predicate `MakeAngle`:

```
(MakeAngle $a $b $c (Angle $a $b $c))
(MakeAngle $a $b $c (Angle $c $b $a))
```

Wherever we want to use `(Angle $a $b $c)` we now say

```
(MakeAngle $a $b $c $abc)
```

and use `$abc` instead. The call to `MakeAngle` succeeds twice if necessary so we can treat `$abc` as if it were an 'unordered' representation of the angle that unifies with either of the ordered versions. Let us see how this works in a simple case, the rule for calculating the third angle of a triangle when the other two are given. The original rule, which is inadequate, is

```
(IF (AND (DegreeValue (Angle $b $a $c) $bacval)
         (DegreeValue (Angle $a $c $b) $acbval)
         (- 180 $bacval $acbval $abcval))
    (DegreeValue (Angle $a $b $c) $abcval)).
```

The first stab at fixing it up is

```
(IF (AND (MakeAngle $a $b $c $abc)
         (MakeAngle $b $a $c $bac)
         (MakeAngle $a $c $b $acb)
         (DegreeValue $bac $bacval)
         (DegreeValue $acb $acbval)
         (- 180 $bacval $acbval $abcval))
    (DegreeValue $abc $abcval))
```

but this rule is actually too general, since we will be normally trying to find some fixed angle `$abc` so we can dispense with that variable and its corresponding `MakeAngle` and use `(Angle $a $b $c)` in the conclusion of the rule, as it appears in the body of the code below. In general, facts such as actual angle values in the database will have a fixed-order representation (we don't want to give

each fact for a problem instance twice), so to achieve generality the premises of rules, which will be unified with facts in the database, should use the MakeAngle method, whilst the conclusions can be in fixed format. If we had the multiple representations of the problem instance we could use MakeAngle for angles in the conclusions of rules and fixed format for those in the premises. The key is to be consistent.

The following rules contain the geometrical theorems that are useful for this proof:

```
;The angle between a tangent and a radius to the point of contact is 90

(IF (AND (MakeLine $a $b $ab)
         (Tangent $ab $circle)
         (Center $circle $o)
         (PointOnCircle $b $circle))
    (DegreeValue (Angle $a $b $o) 90))

;The angles in a triangle add up to 180 (for deducing 3rd angle value)

(IF (AND (MakeAngle $b $a $c $bac)
         (MakeAngle $a $c $b $acb)
         (DegreeValue $bac $bacval)
         (DegreeValue $acb $acbval)
         (- 180 $bacval $acbval $abcval))
    (DegreeValue (Angle $a $b $c) $abcval))

;Angle at the centre is twice the angle at the circumference

(IF (AND (PointOnCircle $a $circle)
         (PointOnCircle $b $circle)
         (Unknown (= $a $b))
         (PointOnCircle $c $circle)
         (Unknown (= $a $c))
         (Unknown (= $c $b))
         (Center $circle $o)
         (MakeAngle $a $o $c $aoc)
         (DegreeValue $aoc $aocval)
         (// $aocval 2 $abcval))
    (DegreeValue (Angle $a $b $c) $abcval))

;The angles at the base of an isosceles triangle are equal

(IF (AND (MakeAngle $b $a $c $bac)
         (MakeLine $b $a $ba)
         (MakeLine $b $c $bc)
         (EqualLength $ba $bc)
         (MakeAngle $a $c $b $acb)
         (DegreeValue $acb $val))
    (DegreeValue $bac $val))
```

```
;All radii of a given circle have equal length

(IF (AND (PointOnCircle $a $circle)
         (PointOnCircle $b $circle)
         (Unknown (= $a $b))
         (Center $circle $o))
    (EqualLength (Line $o $a) (Line $o $b)))

;Angles standing on the same segment are equal

(IF (AND (PointOnCircle $a $circle)
         (PointOnCircle $b $circle)
         (Unknown (= $a $b))
         (PointOnCircle $c $circle)
         (Unknown (= $a $c))
         (Unknown (= $c $b))
         (PointOnCircle $d $circle)
         (Unknown (= $a $d))
         (Unknown (= $b $d))
         (Unknown (= $c $d))
         (MakeAngle $a $d $c $adc)
         (DegreeValue $adc $val))
    (DegreeValue (Angle $a $b $c) $val))

;If P is on AB then <ABC = <PBC

(IF (AND (MakeLine $a $b $ab)
         (PointOnLine $p $ab)
         (MakeAngle $a $b $c $abc)
         (DegreeValue $abc $val))
    (DegreeValue (Angle $p $b $c) $val))

;If A is on PB then <ABC = <PBC

(IF (AND (MakeLine $p $b $pb)
         (PointOnLine $a $pb)
         (MakeAngle $a $b $c $abc)
         (DegreeValue $abc $val))
   (DegreeValue (Angle $p $b $c) $val))

;Angle and line constructors

(MakeAngle $x $y $z (Angle $x $y $z))
(MakeAngle $z $y $x (Angle $x $y $z))
(MakeLine $x $y (Line $x $y))
```

```
(MakeLine $y $x (Line $x $y))
```

The following is the description of the problem instance. These facts should probably be asserted with forward chaining turned on.

```
(PointOnCircle B Q)
(PointOnCircle C Q)
(PointOnCircle D Q)
(PointOnCircle E Q)
(PointOnCircle F Q)
(Center Q O)
(Tangent (Line A B) Q)
(PointOnLine F (Line A O))
(PointOnLine O (Line F D))
(PointOnLine O (Line B E))
(DegreeValue (Angle O A B) 20)
```

Appendix B :

Installation Guide

§B.1 Introduction

There are three different machines upon which this MRS will run. They are *DEC-20*'s running Maclisp, *VAX*en running Franz under Berkeley Unix, and *Symbolics LISP Machine LM-2/3600.* Each machine requires a slightly different set of miscellanous files and slight changes in the file extensions. The package you received has lisp file extensions `.lsp` (DEC-20), `.l` (VAX) or `.lisp` (LM-2/3600). *In the remainder of the document the lisp files will be referred to as* `<file>.lsp`*; the reader should translate this into the appropriate extension for their machine.*

Besides the `lsp`, `test`, `load`, `dat`, `mrs`, `demo` and `doc` files which are used by all the systems, there are several files and file types which are unique to each individual machine. Each is explained in the appropriate section.

Extension	Machine
.CTL	DEC-20
.VAX	VAX
Makefile	VAX
.LM2	LM-2

§B.2 How to get MRS running

B.2.1 Maclisp version on the DEC-20.

Dec-20 tapes are written using the `ANSI` tape program. The files from the tape should be read into the directory in which they will reside on your machine. We maintain the files in the directory `<mrs.mac.cur>`. There are several locations where the directory name is defined. THESE MUST BE UPDATED TO YOUR DIRECTORY NAME. The locations are in all the `CTL` files and in `MRS.LOAD`.

There is an automatic compilation program which will compile all the required lisp programs. It can be run in batch by `"SUBMIT COMPMRS.CTL"`. A summary of the results will be appended to the file `compmrs.log`. If you need to compile individual files, then make sure you are connect to the directory and then run `complr`.

To make an executable version of MRS, run in batch `"SUBMIT MAKMRS.CTL"`. The resulting executable file will reside in `mrs.exe`. Again a log of the results of the batch run is appended to `MAKMRS.CTL`.

You are now ready to start using MRS.

(The file `prmrs.ctl` is used to print out the appropriate files on our laser printer . This is not necessary but is provided as a convenience)

B.2.2 Franz version on the VAX.

Franz-MRS tapes usually are written by the UNIX program `tar`. After you connect to the directory where you want to put the sources, restore them with the command `tar -xp`. (The directory does not have to be dedicated to MRS but this is recommended.)

There are several locations where the directory name is defined. THESE MUST BE UPDATED TO YOUR DIRECTORY NAME. The locations are in `mrs.load` and `Makefile`. The current reference should be to `/hpp/mrs/cur` and should be changed to the directory name.

The file "Makefile" sets the "make" variables LISP (which points to the directory that contains "lisp" and "liszt"), DESTDIR (which points to the directory where the MRS executable file should be linked to), and LIBDIR (which points to the directory where the library files should be linked to). Modify these files for your site.

The file "Makefile" will be used by "make" to build an MRS for you. You must modify the install command so that it sets up the links correctly. (If you don't want the library files to be linked to another directory, this is the place to change.) If you are just interested in a sysout, the command "make xmrs" should put an executable version of MRS into the file "xmrs".

"make install" will create a xmrs in this directory, and also put in a link between the DESTDIR/mrs and the xmrs created in this directory. Only the executable file is linked.

You are now ready to start using MRS.

There are three files with the extension .VAX.

The getfranz.VAX is used to FTP the MRS from our DEC-20 to our VAX. If you have multiple sites running MRS this program can be used. It will have to be altered to your protocols. Notice the name changing from DEC-20 to VAX.

The sendfranz.VAX is used to FTP the MRS from our VAX to our DEC-20. If you have multiple sites running MRS this program can be used. It will have to be altered to your protocols. Notice the name changing from VAX to DEC-20. (getfranz and sendfranz are just opposite directions)

The sendlispm.VAX is used to FTP the code from our VAX to the Symbolics machines. Notice the name changing from VAX to LISPM.

The technique for using getfranz and sendfranz is

```
cat getfranz.VAX | ftp
```

the technique for using sendlispm is:

```
cat sendlispm.VAX | ftp
```

B.2.3 ZetaLisp version on the LM-2/3600.

The lispm-MRS tapes are written on Symbolics streamer tapes. The tape was made using the LMFS dumper and was a complete dump of the MRS directory. The tape can be loaded using the reload/retrieve command on the directory you intend to install MRS. The tape name is MRS-7.1 (or a later version number). Lisp files on the Symbolics machines should end with extension .LISP.

There is one locations where the directory name is defined in the MRS files. THIS MUST BE UPDATED TO YOUR DIRECTORY NAME. The location is in mrs.load. The current reference should be to >mrs>cur> and should be changed to the directory name.

To load MRS into the current lisp environment do the following:

```
(load '>mrs>cur>mrs.load)
```

This should compile the appropriate lisp functions and load them in. There will probably be a

lot of warning messages in the compilation. These on the most part can be ignored. It is advised that MRS be loaded into a clean cold boot. Otherwise names may be redefined.

§B.3 Adding Files to MRS - for system maintainers

If you need to add a new set of source file to MRS, there are several files that are affected and need to be changed. They are `MRS.LOAD` , `Makefile`, `COMPMRS.CTL`, `PRMRS.CTl`, `COPYLISPM.VAX`, `GETFRANZ.VAX`, `SENDFRANZ.VAX`, and `SENDLISPM.VAX`. The files `interface.lsp` and `test.lsp` are affected if new demo packages are added or test files respectively.

In `MRS.LOAD` two references to the lisp source filename (excluding the lisp extension) need to be added

In `Makefile` the source filename (with .l extension) needs to be added to the make variable `SOURCES`, the object filename (with .o extension) needs to be added to the make variable `OBJECTS`, the test filename (with .test extension) needs to be added to the make variable `TESTAUX`, and the demo filename (with .demo extension) needs to be added to the make variable `DEMOAUX`.

In files `COMPMRS.CTL` and `PRMRS.CTL` need to add the source filename (with `.lsp` extension).

In files `COPYLISPM.VAX`, `GETFRANZ.VAX`, `SENDFRANZ.VAX`, and `SENDLISPM.VAX` need to add the source filename (with extension), the test filename (with extension .test) if one exists, and the documentation filename (with extension .doc) if one exists.

If there are any demo files, they need to be added to the list `demolist` in the source file `INTERFACE.LSP`.

If there are any test files, they need to be added to the function `finderrors` in the source file `TEST.LSP`.

§B.4 Testing MRS installation

There is a testing program for verifying the system. Once MRS has been installed call (`finderrors`); it should return a value of 0 indicating that there were no errors encountered. If there are errors in the MRS function being tested, the result and the expected result are indicated.

If you plan on adding `test` files you are required to follow a specific format. The format consists of an MRS (or LISP) command followed by the expected answer. If the answer is irrelevant a * can be used. For examples, look at the `<file>.test` source files.

§B.5 The Share Subdirectory

We are also maintaining a subdirectory called `share` which is composed of user written code that may be useful to other sites. These procedures are *not* currently considered a core part of MRS and hence are not in the main directory. All files maintained in this subdirectory should have a `*.doc` file describing the use and functions in the file and `*.dict` file consisting of a dictionary entry for each of the user accessable functions/procedures/relations.

§B.6 Required Files

This is a list of the required files that should be on your tape. There may be additional files which can be ignored.

LISP files

```
ask.lsp
base.lsp
batch.lsp
bc.lsp
cnf.lsp
common.lsp
compat.lsp
erfrepn.lsp
execute.lsp
fc.lsp
interface.lsp
macros.lsp
match.lsp
meta.lsp
mla.lsp
plist.lsp
proprep.lsp
repn.lsp
res.lsp
set.lsp
test.lsp
timer.lsp
tm.lsp
top.lsp
toplevel.lsp
tr.lsp
trbc.lsp
trexec.lsp
trfc.lsp
tut-concpt.lsp
tut-dict.lsp
tut-exer.lsp
tut-gen.lsp
tut-main.lsp
tut-synchk.lsp
tut-topics.lsp
version.lsp
```

Demo files

```
blocks.demo
d74.demo
kinship.demo
overview.demo
primate.demo
tax.demo
```

Test files

```
base.test
bc.test
erfrepn.test
execute.test
fi.test
fc.test
meta.test
mla.test
proprep.test
repn.test
res.test
set.test
tr.test
trbc.test
trfc.test
```

MRS files used by demo, test

```
blocks.mrs
fi.mrs
kinship.mrs
meta.mrs
mrs.mrs
msai.mrs
primate.mrs
s.mrs
sets.mrs
syntax.mrs
tax.mrs
```

VAX files

```
Makefile
ReadMe
copylispm.vax
getfranz.vax
sendfranz.vax
sendlispm.vax
```

DEC-20 files

```
compmrs.ctl
makmrs.ctl
prmrs.ctl
```

Appendix C :

Dictionary of predicates and flags

***** (* <a_1> ... <a_n> <a_{n+1}>)
means that <a_{n+1}> represents the product of the values represented by <a_1>...<a_n>. See fq.

+ (+ <a_1> ... <a_n> <a_{n+1}>)
means that <a_{n+1}> represents the sum of the values represented by <a_1>...<a_n>. See fq.

- (- <a> <b> <c>)
means that <c> represents the difference between <a> and <b>. See fq.

// (// <a> <b> <c>)
means that <c> represents the quotient of <a> and <b>. See fq.

< (< <a> <b>)
means that <a> is less than <b>. See rq.

<= (<= <a> <b>)
means that <a> is less than or equal to <b>. See rq.

= (= <a> <b>)
means that the terms <a> and <b> are synonymous, i.e. they refer to the same object. See lookup-=.

> (> <a> <b>)
means that <a> is greater than <b>. See rq.

>= (>= <a> <b>)
means that <a> is greater than or equal to <b>. See rq.

achieve (achieve <p>)
makes the proposition <p> true. Achieve is an abstract operator implemented using kb and toachieve. (achieve <p>) and (achieve (not <p>)) now work when the proposition <p> begins with value, property, repn, threpn, includes, indb, better, or primitive. So, up to a point, does (achieve (if <q> <p>)), which calls trueps on <q> and then ACHIEVEs <p> with the resulting bindings plugged in. Note that propositions stashed in theories other than the currently writeable one are not affected. (e.g. (achieve (repn <p> <r>)) has the result that all propositions matching <p> will henceforth be stashed, lookuped, and so on using representation <r>. This includes re-storing currently accessible propositions that were stored using pr-stash or in cnf. repn works on all theories that are active when it is achieved. (achieve (threpn <p> <r> <th>))) does the same thing as with repn, but affecting only theory <th>. It is the users responsibility to ensure that there are never two or more theories active which use different representations for the same proposition.) See kb, repn, threpn.

achieve-if (achieve-if (if <p> <q>))
has the effect of calling achieve on the proposition <q> for each list of variable bindings that makes proposition <p> true, with the relevant

bindings substituted into <q>. See achieve, trueps, plug.

achieve-not (achieve-not (not <p>))
is an abstract operator implemented using kb and tounachieve. When called with an argument of the form (not <p>), it is supposed to achieve the opposite of <p>, if meaningful. See achieve, unachieve.

achieve-repn (achieve-repn (repn <prop> <rpn>))
uses repn-assert to switch the representation of <prop> from its old value to <rpn>, and converts any instances of <prop> that can be found under the old representation to the new one. See domain, repn, repn-assert, repn-method.

achieve-threpn (achieve-threpn (repn <prop> <rpn> <th>))
is identical to achieve-repn except that it sets the currently writable theory to <th> temporarily while it executes. See achieve-repn, theory, threpn.

activate (activate $<t_1> \ldots <t_n>$)
makes the propositions in the theories $<t_1> \ldots <t_n>$ available for retrieval or deduction. See theory, activetheories, deactivate.

activetheories has as its value the list of currently active theories. The propositions in these theories are available for retrieval by pr-lookup and pr-lookups. See pr-stash, pr-unstash, pr-lookup, pr-lookups, activate, and deactivate.

agenda agenda is a list of applicable tasks. See applicable, scheduler.

and (and $<p_1> \ldots <p_n>$)
means that the propositions $<p_1> \ldots <p_n>$ are all true. See assert-and, bc, br, fc.

applicable (applicable <k>)
states that the task <k> is applicable and, therefore, executable unless it is disqualified. See executable, disqualified, scheduler.

arity (arity <rel> <i>)
provides typing information, indicating that the relation (or operation) <rel> takes <i> arguments, e.g. (arity arity 2). See domain.

ask (ask <p>)
calls output, prints the result, and reads the users answer. If <p> is a ground proposition, ask tries to obtain an answer of true or false. If <p> contains variables, ask obtains variable bindings from the user. See output.

asks (asks <p>)
calls output, prints the result, and reads the users answers. If <p> is a ground proposition, asks tries to obtain an answer of true or false. If <p> contains variables, asks obtains a list of binding lists from the user. See output.

assert **(assert <p>)**
stores the proposition <p> in the data base and performs all appropriate forward inference. **Assert** is an abstract operator implemented using **kb** and **toassert**.

assert-and **(assert-and (and $<p_1>$ ... $<p_n>$))**
separately asserts each of the conjuncts $<p_1>$,..., $<p_n>$.

assert-iff **(assert-iff (iff <p> <q>))**
asserts **(if <p> <q>)** and **(if <q> <p>)**.

assumable **(assumable <p>)**
means that the proposition <p> can be assumed if necessary in trying to prove a proposition. See **residue**, **residues**.

bagof **(bagof <x> <p> <s>)**
means that <s> is the bag of all objects <x> that statisfy <p>. Since there maybe many ways of satisfying <p>, the bag <s> may contain duplicate objects. **Bagof** is useful for performing extensional reasoning, since it allows one to designate the set of all solutions to a problem. See **truep-bagof**, **lookup-bagof**.

batchp **(batchp <x> <y>)**
checks whether the expressions <x> and <y> can be unified by some set of bindings for the base-level variables in the two expressions. If so, **batchp** returns the corresponding binding list for the variables in <x> but discards the bindings for the variables in <y>. If the expressions are not unifiable, **batchp** returns **nil**. All variables in <x> are treated as distinct from the variables in <y>, even though they have the same name. For example, the expression **(r $x b)** matches **(r a $x)** with result **(($x . a) (t . t))**. See **blvarp**, **matchp**.

bc **(bc <p>)**
tries to prove the proposition <p>. If successful, it returns an appropriate binding list; otherwise, it returns **nil**. Only base-level variables are treated as variables by **bc**, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the **totruep** and **totrueps** relations). **Bc** is implemented using the subroutine **bcdisp**. See **bcdisp**, **scheduler**.

bcdisp **(bcdisp <gl> <al> <jl> <ce>)**
performs one backward chaining step in trying to prove the propositions on the goal list <gl>. The binding list <al> holds bindings for the variables in <gl> obtained in preceding steps. The justification list <jl> holds the names of any propositions used in deriving a goal list from its super goal list. The list <ce> is a stack of supergoals and justifications. In working on a goal list **(<q> . <l>)**, **bcdisp** first calls **trtruep** to find any procedural attachments for <q> other than **bc** or **bcs**, and if successful calls that subroutine. Otherwise, it generates subgoals by looking in the data base for propositions of the form <q> or **(if <p>**

<q>). The order in which multiple bcdisp tasks are executed can be influenced via appropriate preferred propositions. bcdisp caches its results and saves justifications as appropriate. See trtruep, totruep, totrueps, cache, justify.

bcs (bcs <p>)
tries to prove the proposition <p>. It returns a list of all binding lists for which it is successful. Only base-level variables are treated as variables by bcs, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the totruep and totrueps relations). Bcs is implemented using the subroutine bcdisp. See bcdisp and scheduler.

blvarp (blvarp <xp>)
returns a non-nil value if <xp> is a base-level variable and otherwise returns nil. A base-variable in MRS is denoted by a dollarsign prefix ($) and is internally distinguished by the value bl. For example, $a is a base-level variable. See varp.

br (br <p>)
tries to prove the proposition <p>. If succesful, it returns a list of assumable propositons which, when added to the data base, imply <p>. Only base-level variables are treated as variables by br, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the totruep and totrueps relations). Br is implemented using the subroutine brdisp. See brdisp, scheduler, and assumable.

brdisp (brdisp <gl> <al> <th> <jl> <ce>)
performs one backward chaining step in trying to find a residue for the propositions on the goal list <gl>. The binding list <al> holds bindings for the variables in <gl> obtained in preceding steps. The theory <th> contains all assumptions made so far. The justification list <jl> holds the names of any propositions used in deriving a goal list from its super goal list. The list <ce> is a stack of supergoals and justifications. In working on a goal list (<q> . <l>), brdisp first calls trtruep to find any procedural attachments for <q> other than bc or bcs, and if successful calls the subroutine so found. Otherwise, it generates subgoals by looking in the data base for propositions of the form <q> or (if <p> <q>). It uses trueps to discover whether q is assumable. If it is assumable and if is a ground proposition after plugging in the variable bindings returned by trueps, brdisp creates a new theory that includes <th>, asserts the proposition in that theory, and generates appropriate subgoals. The asserted propositions are useful in that they make possible consistency checking before making assumptions in subsequent steps. The order in which multiple brdisp tasks are executed can be influenced via appropriate preferred propositions. Brdisp caches its results and

saves justifications as appropriate. See trtruep, totruep, totrueps, cache, justify.

brs (brs <p>)
tries to prove the proposition <p>. It returns a list of all assumption lists for which it is successful. Only base-level variables are treated as variables by brs, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the totruep and totrueps relations). Brs is implemented using the subroutine brdisp. See brdisp, scheduler, and assumable.

cache is a variable governing whether various inference methods should cache their results. When nonNIL, those various inference methods will call the appropriate tocache method on each cachable result. (I.e., each method will call (kb tocache <p>) for each intermediate conclusion, <p>.) See tocache, cachebystash.

cachebystash (cachebystash <p>)
stashes the value <p> into the theory named by the variable cache. (If cache has the value T, then the current theory is used.) This cachebystash subroutine is the default caching method. It is recommended that one house these propositions in a temporary theory, and apply empty to this theory when the cached values are no longer needed. See cache, tocache

characteristic (characteristic <set> <fn>)
means that the lisp subroutine <fn> is the characteristic function for the set <set>, e.g. (characteristic integers fixp). See arity, domain.

cnf is short for conjunctive normal form. A proposition is in conjunctive form if it is written as a conjunction of disjunctions of literals, i.e. atomic propositions or negations of atomic propositions. For example, the proposition (and (or (not (p $x)) (q $x)) (or (r $x) (s $x))) is in conjunctive form. This also serves as a representation. See repn.

cnf-assert (cnf-assert <p>)
converts <p> into conjunctive normal form and separately asserts each of the conjuncts. See cnf.

cnf-unassert (cnf-unassert <p>)
converts <p> into conjunctive form and separately unasserts each of the conjuncts. See cnf.

computable-repn Explanation.
Many relations and functions can be readily evaluated, and so never need to be explicitly stashed. Consider, for example, the class of arithmetic functions and relations, such as + and >. MRS includes several computable representations in which to encode such facts. We describe

below various computable representations – viz., re, rq, rqb, rqbm, reb, rebm, rqfm, fe, fq and fea. For a proposition to be represented in one of these representation, its relation symbol must have an associated LISP subroutine (ALS). Each lookup subroutine associated with each of these representations takes as input a proposition of the form (<r> $<x_1>$ ... $<x_n>$), and calls ALS on a list of values computed from that argument list, $<x_1>$ ... $<x_n>$. In some representations, each argument is first evaluated, using lookupval. Also, in some representations the ALS takes all n arguments, while in others it is only passed the term-part of the proposition, namely the first n-1 arguments. Note that propositions stored in this way are not associated with any particular theory and cannct be found by PR-based routines like prfacts or prcontents. The details of these representations are specified in the Computable-Repn (Relation) and Computable-Repn (Function) entries. These representations are used by the funproc and relnproc relations.

computable-repn Functions.

Here we describe those computable-repn representations which are based on a function. We designate these function-based representations by using the letter F in the first position of the name of the representation – e.g. Fq is a function-based representation. Here, only the term part of the proposition is passed to the ALS; and it is the responsibility of the associated lookup subroutines to bind this returned value appropriately. The same second letter is E for Eval, Q for Quote convention used for the relation-based representations applies here as well. Hence lookup subroutines associated with the FE representation will first lookupval each of the arguments $<x_1>$ ... $<x_{n-1}>$, passing the resulting list to the ALS. The only (current) additional letter for function-based representations is A, for arithmetic. This uses NUM-= rather than UNIFYP when comparing the value associated with the term of the proposition with the value of the proposition. See computable-repn, fe, fea, fq.

computable-repn Relations.

This section describes those computable-repn which representations are based on relations. These relation-based representations are designated by using the letter R in the first position of the name of the represenation – e.g. Erq is a relation-based representation. With one exception, the ALS takes a spread version of the full proposition as its arguments. If the second letter is E for eval, (as in rEbm,) the associated lookup methods will first call lookupval on each embedded term, and pass that evaluated argument list to the ALS. Otherwise, when it is Q for quote, those argument are directly passed to the ALS. By default, the value returned by the ALS is an arbitrary value which, when nonNIL, tells the lookup subroutine that this propostion is true. The third and fourth letter encode further refinements: When the third letter in the representations name is B, (e.g. reB,) the ALS itself will return a binding list, which the lookup subroutine will return. For these r?b relations, a subsequent

M (e.g. rebM,) means the ALS returns a list of binding-lists, rather than just one. This 4th letter, M, means multiple values convention is retained for the RQFM representation (used for multi-functions). Here the ALS takes only the term part of the proposition, and returns a set of values. The RQFM-LOOKUPS subroutine then forms the list of appropriate binding-lists. Consider the Square-Root multi-function, which returns both the + and - root of a number. See Computable-Repn (Concept), repn, re, reb, rebm, rq, rqb, rqbm, rqfm.

contents (contents <t>)
returns a list of propositions stored in theory <t>. Only those facts stored using the propositional representation (i.e., pr) will be found.

cut (cut)
is a special control form. When (cut) is executed, all other subtasks of the enclosing doable or undoable task are discarded. As a result, if the subtask containing the (cut) form fails, the enclosing doable or undoable subtask will fail as well. See doable, undoable.

datum (datum <x>)
returns the symbol corresponding to the expression or proposition <x>.

deactivate (deactivate $<t_1> \ldots <t_n>$)
deactivates the named theories. See theory, activetheories, activate.

def (def <k> <k1> ... <kn>)
means that the task <k> is defined as (doand <k1> ... <kn>). A task can have more than one definition, each one covering a different set of inputs. For example, the following propositions define the factorial function.

```
(def (fact 0 1))
(def (fact &m &n)
    (- &m 1 &p)
    (fact &p &q)
    (* &m &q &n))
```

defobject (defobject <name> $<p_1> \ldots <p_n>$)
unasserts all propositions in pr that mention <name> and then asserts the propositions $<p_1>, \ldots, <p_n>$.

defrule (defrule <rule> $<f_1> \ldots <f_N>$)
asserts the rule <rule>, and then trasserts each meta-level $<f_i>$, after substituting the symbol associated with <rule> for *. A typical application is (defrule '(if (a $x) (b $x)) '(direction * forward)), which asserts (if (a $x) (b $x)) and trasserts (direction p307 forward), where p307 is the proposition symbol assigned to (if (a $x) (b $x)). As a shorthand, the * may be omitted for unary functions like direction. That is, '(direction forward) could be used in place of '(direction * forward). See direction and

trassert.

`deftheory` (deftheory <name> <p_1 > ... <p_n > empties the theory <name> and asserts propositions <p_1 >,..., <p_n > into it.

`direction` (direction <p> <d>)
means that the rule whose symbol is <p> should only be used in the <d> direction, where <d> is forward, backward or both. E.g., after (trstash '(direction p307 forward)), the rule named by p307 – say, (if (a $x) (b $x)) – will only be used in the forward direction. That is, the fc subroutines will be able to use this rule in forward-chaining (e.g. from (a 19)), but neither bc nor br will have access to this rule. This is implemented via the direction-lookup, direction-stash and direction-unstash subroutines. By default, all rules can be used in both directions. See bc, br, defrule and fc.

`disjoint` (disjoint <x> <y>)
means that lists <x> and <y> do not have any elements in common.

```
(disjoint nil $y)
(disjoint $x nil)
(if (and (not (element $e $s)) (disjoint $l $s))
    (disjoint ($e . $l) $s))
```

Procedural attachment: truep-disjoint. The lisp file set must be loaded from the mrs directory.

`disqualified` (disqualified <k>)
states that the task <k> is disqualified. A task that is applicable is executable unless it is disqualified. The chief way a task can get disqualified is for there to be another applicable task that is preferred to it. However, this fact is used when either of the switches executable or preferred is non-nil. See applicable, scheduler.

`dl` (repn <p> dl)
means that the proposition <p> should be represented in the dl representation, i.e. the dl-<x> family of subroutines will be used to stash, unstash, and lookup <p>. This representation is particularly useful for representing propositions involving non-functional binary relations, e.g. (neighbor france switzerland). Note that propositions stored in this way are not associated with any particular theory and cannot be found by PR-based routines like prfacts or prcontents. See dl-lookup, dl-stash, dl-unstash, repn.

`dl-lookup` (dl-lookup (<r> <a> <b>))
matches <b> against each of the values stored as the <r> property of the lisp atom <a>, and if successful returns the resulting binding list. Both <r> and <a> must be atoms. See dl.

`dl-lookups` (dl-lookups (<r> <a> <b>))
matches <b> against each of the values stored as the <r> property of the lisp atom <a>, and returns a list of the binding lists for every successful match. Both <r> and <a> must be atoms. See dl.

dl-stash (dl-stash (<r> <a> <b>))
adds <b> to the list of values stored as the <r> property of the atom <a>. Both <r> and <a> must be atoms. See dl.

dl-unstash (dl-unstash (<r> <a> <b>))
removes <b> from the list of values stored as the <r> property of <a>. Both <r> and <a> must be atoms. See dl.

dnf is short for disjunctive form. A proposition is in disjunctive form if it is written as a disjunction of conjunctions of literals, i.e. atomic propositions or negations of atomic propositions. For example, the proposition (or (and a b) (and c d)) is in disjunctive form. This also serves as a representation. See repn.

doable (doable <k>)
designates the task of trying to execute the task <k>. The task (doable <k>) succeeds if there is a successful execution of <k>. However, after a single success, all other subtasks of <k> are discarded, and so it can succeed at most once. Note that as a result of the current implementation, it is not possible to interleave subtasks outside a doable task with those inside. See succeed and cut.

doall (doall <x> <k> <s>)
designates the task of getting all <x> for which the task <k> succeeds. It packages these into a list and succeeds if the resulting list unifies with <s>.

doand (doand $<k_1> \ldots <k_n>$)
designates the task of executing tasks $<k_1>, \ldots, <k_n>$ in sequence. If one of the tasks can be executed in more than one way, separate subtasks are set up for each possibility. The doand task succeeds if there is at least one succesful execution of all the tasks in the list. For example, (doand (memlist &x ((a) b (c))) (atom &x t)) succeeds because the atom task succeeds for the execution of the mem task in which &x is bound to b. The order in which subtasks are executed can be influenced by making appropriate preferred statements.

domain (domain <rel> <i> <set>)
provide typing information, indicating that the <i>th argument of the relation (or operation) <rel> must belong to the set <set>. E.g. (domain stash 1 propositions). See arity, charactistic, theories, terms, propositions.

door (door $<k_1> \ldots <k_n>$)
designates the task of trying to execute one the tasks $<k_1>, \ldots, <k_n>$.

edunit (edunit <x>) allows the user to edit the propositions about <x> using Zwei. AVAILABLE ONLY IN ZETALISP.

element (element <x> <s>)
means that object <x> is an element of list <s>.

```
(element $e ($e . $1))
(if (element $e $1) (element $e ($y . $1)))
```

Procedural attachment: truep-element. The lisp file set must be loaded from the mrs directory.

elementsin (elementsin <b> <s>)
means that <s> is the set of all elements in the bag <b>.

```
(elementsin nil nil)
(if (and (not (element $e $1)) (elementsin $1 $s))
    (elementsin ($e . $1) ($e . $s)))
(if (and (element $e $1) (elementsin $1 $s))
    (elementsin ($e . $1) $s))
```

Procedural attachment: truep-elementsin. The lisp file set must be loaded from the mrs directory.

empty (empty <t>)
unasserts all the facts in the theory <t>. Only those facts stored using the propositional representation (i.e., pr) will be found.

exdisp (exdisp <l> <al>)
performs one step in the execution of the list of tasks <l>. The alist <al> is a list of variable bindings obtained so far.

executable (executable <k>)
states that the task <k> is executable. If the value of the variable executable is non-nil, scheduler uses trtruep to find a task <k> such that (executable <k>) is true. If the value of executable is nil, it simply selects an element from the value of the variable agenda. If the value of the variable preferred is nil, it takes the first element; otherwise, it uses trtruep to find the best according to the preferred relation. See scheduler.

executable See definition of the meta-level executable

execute (execute <k>)
tries to execute the task <k> and, if successful, returns a binding list for the meta-level variables in <k>. It is implemented using exdisp. See task, def, doable, undoable, doall, doand, door, succeed, cut, preferred, tracetask.

executed (executed <k>)
means that the task <k> has been executed. If the value of the variable executed is non-nil, scheduler will use trassert to record this fact of each task as it is performed. See scheduler.

executed See definition of the meta-level executed.

executes (executes <p>)
tries to execute the task <k> and, if successful, returns a list of all binding lists for the meta-level variables in <k>. It is implemented using exdisp. See task, def, doable, undoable, doall, doand, door, succeed, cut, preferred, tracetask.

fc **(fc <p>)**
means that proposition <p> is assert and then all rules with <p> in the premise are checked to see if the premise is true and if so the consequence is asserted. Only base-level variables are treated as variables by fc, and any meta-level variables are treated as constants. The inference procedure used is forward chaining, but there are also built-in procedural attachments for many propositions (specified via the toassert relation). Fc is implemented using the subroutine fcdisp. See fcdisp, scheduler.

fcdisp **(fcdisp <p>)**
performs one forward chaining step on the proposition <p>. Fcdisp first calls trtruep to find a procedural attachments for <p> other than fc, and if successful calls that subroutine. Otherwise, it generates other assertions by looking in the data base for propositions of the form (if <p> <q>) and (if (and ... <p> ...) <q>). The order in which multiple fcdisp tasks are executed can be influenced via appropriate preferred propositions. Fcdisp caches all results and saves justifications as appropriate. See trtruep, toassert, justify.

fe **(repn <p> fe)**
means that the proposition <p> should be represented in the fe representation. That is, the fe-<x> family of subroutines will be used to retrieve <p>. Its lookup method, fe-lookup, takes as its argument a proposition of the form (<f> $<x_1>$ ... $<x_n>$ $<x_{n+1}>$). It first evaluates the term (<f> $<x_1>$ ... $<x_n>$) by applying LISP subroutine corresponding to the function symbol <f>, (i.e., on <f>s LISP property,) to the list obtained by calling lookupval on each of the $<x_i>$. Fe-Lookup then unifies this value with (lookupval $<x_{n+1}>$), and returns the result. See computable-repn, fe-lookup, fe-lookups, fea, fq, funproc, repn.

fe-lookup **(fe-lookup <p>)**
is used to retrieve the proposition <p>. See fe, lisp.

fe-lookups **(fe-lookups <p>)**
is functionally equivalent to (pluralize (Fe-lookup p)). See fe, fe-lookup.

fea **(repn <p> fea)**
means that the proposition <p> should be represented in the fea representation. That is, the fea-<x> family of subroutines will be used to retrieve <p>. Its lookup method, fea-lookup, takes as its argument a proposition of the form (<f> $<x_1>$ ... $<x_n>$ $<x_{n+1}>$). It first evaluates the term (<f> $<x_1>$ ... $<x_n>$) by applying LISP subroutine corresponding to the function symbol <f>, (i.e., on <f>s LISP property,) to the list obtained by calling lookupval on each of the $<x_i>$. Fea-Lookup then compares this value to (lookupval $<x_{n+1}>$) using num-=, and returns the result. (Fe-lookup just uses unifyp to produce this comparison.) See computable-repn, fea-lookup, fea-lookups, fe, funproc, repn.

fea-lookup (fea-lookup <p>)
is used to try to retrieve the proposition <p>. See fea, num-=, lisp.

fea-lookups (fea-lookups <p>)
is functionally equivalent to (pluralize (Fea-lookup p)). See fea, fea-lookup.

finderrors (finderrors)
runs through all the test files to find the errors in MRS. For each error it find it prints out the result and expected result. It returns the number of errors it finds

fq (repn <p> fq)
means that the proposition <p> should be represented in the fq representation. That is, the fq-<x> family of subroutines will be used to retrieve <p>. By default, most functions, (including arithmetic ones,) use this representation. Its lookup method, fq-lookup, takes as its argument a proposition of the form (<f> $<x_1>$ ... $<x_n>$ $<x_{n+1}>$). It first evaluates the term (<f> $<x_1>$ ... $<x_n>$) by applying LISP subroutine corresponding to the function symbol <f>, (i.e., on <f>s LISP property,) to the list $<x_1>$ ... $<x_n>$. Fq-Lookup then unifies this value with $<x_{n+1}>$, and returns the result. See computable-repn, fe, fq-lookup, fq-lookups, fqm, repn.

fq-lookup (fq-lookup <p>)
is used to try to retrieve the proposition <p>. See fq lisp.

fq-lookups (fq-lookups <p>)
is functionally equivalent to (pluralize (Fq-lookup p)). See fq, fq-lookup.

function (function <x>)
means that <x> is a function. See singlep

funproc (2) (funproc <sym> <op>)
means that the operator <op> is procedurally attached to the symbol <sym>. E.g. after asserting (funproc + plus), (lookupval (+ 2 3 4)) will pass 2, 3 and 4 to the LISP procedure plus, and return its answer, 9. Note that this looking-up mechanism is (intentionally) very simple – while (lookup (+ 2 3 4 $a)) will work (returning a binding list which includes ($a . 9),) both (lookup (+ 2 3 $b 9)) and (lookup (+ $d 0 $c)) will return nil. The (funproc <sym> <op>) assertion will use funproc-assert to forward chain to assert (function <sym>), and that all propositions (and terms) whose relation symbol is <sym> should use the fq representation. See fq, function, funproc (3), funproc-assert, relnproc.

funproc (3) (funproc <sym> <op> <repn>)
This is an embellishment of the binary funproc, listed under funproc (2). After a (funproc + plus) assertion, both (lookup (+ 2 (+ 3 4) $x)) and (lookup (+ 2 3 4) 9.0)) will fail – i.e. return nil. One

can use the (optional) third argument, <repn>, to permit two other, more elaborate forms of functional procedural attachment. The (funproc + plus FE) assertion handles the first problem. Now each embedded ground term – here 2 and (+ 2 3) – will first be lookupvaled, and the result passed to the LISP procedure plus. That is, this uses the fe representation, rather than fq. The assertion (funproc + plus FEA) solves the second problem, causing (+ . &x) to use the fea representation. The <repn> term defaults to fq if omitted. One can also substitute EVAL for fe, or = for fea. See fea, fe, fq, function, funproc (2), funproc-assert, num-=, relnproc.

funproc-assert (funproc-assert (funproc <sym> <op> <repn>))
asserts the proposition (funproc <sym> <op> <repn>). (The same subroutine is used to assert (funproc <sym> <op>).) See funproc.

getbdg (getbdg <v> <p>)
is equivalent to (getvar <v> (truep <p>)).

getbdgs (getbdgs <v> <p>)
is equivalent to
(mapcar (lambda (l) (getvar <v> l)) (trueps p)).

getval (getval (<r> $<x_1>$ ... $<x_n>$))
is equivalent to (getbdg <y> (<r> $<x_1>$ ... $<x_n>$ <y>)).

getvals (getvals (<r> $<x_1>$ ... $<x_n>$))
is equivalent to (getbdgs <y> (<r> $<x_1>$ ... $<x_n>$ <y>)).

getvar (getvar <v> <l>)
looks up the binding of the variable <v> on the binding list <l>, fully instantiates it with respect to the other variables on <l>, and returns the result. For example, (getvar $x (($x . (f $y)) ($y . a))) would return (f a).

ground (ground <x>)
states that the expression <x> is a ground expression, i.e. it contains no variables. See lookup-ground.

groundp (groundp <x>)
returns t if and only if the expression <x> contains no variables.

if (if <p> <q>)
means that whenever proposition <p> is true, proposition <q> is true. See bc, br, fc.

iff (iff <p> <q>)
is equivalent to (and (if <p> <q>) (if <q> <p>)).

includes (includes <c> <d>)
means that the theory <c> includes the theory <d>.

includes (includes $<t_1>$ $<t_2>$)
makes theory $<t_1>$ a supertheory of theory $<t_2>$ so that whenever

<t_1 > is active <t_2 > will be active as well. In effect theory <t_1 > includes all of the propositions in <t_2 >. Both theory and activetheories take these inclusions into account.

indb (indb <p>)

indbp (indbp <p>)
means that a proposition equal to <p> up to variable renaming is stored in the pr representation. See pr-indbp.

integer (integer <x>)
means that <x> is an integer.

inter (inter <x> <y> <b>)
means that list <b> is every element in list <x> that is in list <y>.

```
(inter nil $y nil)
(if (and (element $e $y) (inter $l $y $s))
    (inter ($e . $l) $y ($e . $s)))
(if (and (not (element $e $y)) (inter $l $y $s))
    (inter ($e . $l) $y $s))
```

Procedural attachment: truep-inter. The lisp file set must be loaded from the mrs directory.

intersect (intersect <x> <y>)
means that bag <x> and bag <y> have an equal element.

```
(if (or (element $e $s) (intersect $l $s))
    (intersect ($e . $l) $s))
```

Procedural attachment: truep-intersect. The lisp file set must be loaded from the mrs directory.

is (is <x> <y>)
means that the value of the arbitrarily nested expression <x> is <y>. See lookup-is, truep-is.

just (just <q> <m> <p_1 > ... <p_n >)
means that the justification for the proposition named <q> is the inference method <m> and the premises <p_1 >, ... <p_n >. See where, why, justify, tm-unassert.

justify is a variable governing MRSs mechanism for recording justifications. When nonNIL, its value is the name of the theory into which MRS will save justifications for all deductions. (If justify has the value T, then the current theory is used.) It is recommended that one house these propositions in a temporary theory, and apply empty to this theory when these justifications are no longer needed. See why and where.

kb (kb to<g> <arg_1 > ... <arg_N >)
is MRSs way of handling procedural attachments. It is equivalent to (apply (getvar &f (trtruep (to<g> <arg_1 > ... <arg_N > &f) <arg_1 > ... <arg_N >)). See to<g>.

known (known <p>)
means that proposition <p> must be in the database for this to be true.

See unknown.

length (length <l> <n>)
means that the list <l> is of length <n>.

lhfalse (lhfalse (unprovable <p>))
calls lookup on the proposition <p>. It returns nil if the answer is non-nil; otherwise, it returns truth.

lhtrue (lhtrue (provable <p>))
calls lookup on the proposition <p> and returns the answer.

lisp (lisp <sym> <op>)
means that <op> is the Lisp subroutine used to compute the function denoted by the symbol <sym>. E.g. (lisp + plus). See Computable-Repn (Concept), fea-lookup, fe-lookup, fq-lookup, re-lookup, rq-lookup, rqb-lookup, reb-lookup, rqfm-lookups.

lookup (lookup <p>)
checks whether the proposition <p> matches a proposition in the data base and, if so, returns the correponding binding list. Lookup is an abstract operator implemented using kb and tolookup.

lookup-= (lookup-= (= <x> <y>))
calls unifyp on the expressions <x> and <y> and returns the result. See =.

lookup-bagof (lookup-bagof (bagof <x> <p> <s>))
calls lookups on <p> and matches <s> against the sequence formed by plugging the answers into <x>. Lookup-bagof is useful for performing extensional reasoning, since it allows one to designate the set of all solutions to a problem.

lookup-ground (lookup-ground (ground <x>))
returns ((t p t)) if and only if the expression <x> contains no variables. See ground.

lookup-is (lookup-is (is <x> <y>))
uses lookupval to evaluate the arbitrarily nested expression <x> and tries to unify the answer with <y>. See is.

lookupapplicable (lookupapplicable (applicable <k>))
tries to match <k> with each element of agenda and returns a corresponding binding list if successful. See applicable.

lookupbdg (lookupbdg <v> <p>)
is equivalent to (getvar <v> (lookup <p>)).

lookupbdgs (lookupbdgs <v> <p>)
is equivalent to (mapcar (lambda (x) (getvar <v> x)) (lookups <p>)).

lookupbylookups (lookupbylookups <p>)
is equivalent to (singularize (lookups <p>)).

`lookups` `(lookups <p>)`
checks whether the proposition <p> matches any propositions in the data base and returns a list of binding lists for each successful match. Lookups is an abstract operator implemented using kb and tolookups.

`lookupsapplicable` `(lookupsapplicable (applicable <k>))`
tries to match <k> with each element of agenda and returns list of binding lists for each successful match. See applicable.

`lookupsbylookup` `(lookupsbylookup <p>)`
is equivalent to (pluralize (lookup <p>)).

`lookupval` `(lookupval (<f> <x1> ... <xn>))`
is equivalent to (getvar <y> (lookup (<f> <x1> ... <xn> <y>))).

`lookupvals` `(lookupvals (<f> <x1> ... <xn>))`
is equivalent to (mapcar (lambda (x) (getvar <y> x)) (lookups (<f> <x1> ... <xn> <y>))).

`mand` `(mand <p> <l>)`
is satisfied if (<p> x) is true for every x in the list <l>.

```
(mand $p nil)
(if (and ($p $x) (mand $p $l))
    (mand $p ($x . $l)))
```

Procedural attachment: truep-mand. The lisp file set must be loaded from the mrs directory.

`mandcan` `(mandcan <p> <l> <s>)`
means that <s> is the union of the lists y that satisfy (<p> x y) for every element x in list <l>.

```
(mandcan $f nil nil)
(if (and ($f $x $y) (mandcan $f $l $s) (union $y $s $t))
    (mandcan $f ($x . $l) $t))
```

Procedural attachment: truep-mandcan. The lisp file set must be loaded from the mrs directory.

`mandcar` `(mandcar <p> <l> <s>)`
means that <s> is the set of objects y that satisfy (<p> x y) for every element x in list <l>.

```
(mandcar $f nil nil)
(if (and ($f $x $y) (mandcar $f $l $s))
    (mandcar $f ($x . $l) ($y . $s)))
```

Procedural attachment: truep-mapcar. The lisp file set must be loaded from the mrs directory.

`matchp` `(matchp <x> <y>)`
checks whether the expressions <x> and <y> can be unified by some set of bindings for the meta-level variables in the two expressions. If so, matchp returns the corresponding binding list for the variables in <x> but discards the bindings for the variables in <y>. If the expressions are not unifiable, matchp returns nil. All variables in <x> are treated

as distinct from the variables in <y>, even though they have the same name. For example, the expression (r &x b) matches (r a &x) with result ((&x . a) (t p t)). See batchp.

mem (mem <e> <c>)
means that the element <e> is a member of the set of <c>. E.g., (mem george people). See subclass.

member (member <a> <s>)
means that <a> is a member of the list <s>, e.g. (member 5 (4 5 6)).

memlist (memlist <x> <l>)
designates the task of checking whether the object <x> is in the list <l>. Memlist tries to unify <x> with each element of <l> and succeeds once for each match that it finds.

mlvarp (mlvarp <xp>)
returns a non-nil value if <xp> is a meta-level variable and otherwise returns nil. A meta-variable in MRS is denoted by an ampersand prefix (&) and is internally distinguished by the value ml, e.g. &a is a meta-level variable. See varp.

mrsapropos (mrsapropos <s>)
returns a list of all LISP atoms containing <s> as a substring of their printnames.

mrsdemo (mrsdemo)
presents a demonstration of the range of capabilities in MRS.

mrsdescribe (mrsdescribe <x>)
prints out the portion of this dictionary relevant to the object <x>.

mrsdump (mrsdump <t> <f>)
saves the propositions from theory <t> in a form that allows them to be reloaded with LISPs load command. Only those facts stored using the propositional representation (i.e., pr) will be found.

mrshelp (mrshelp <k>) provides information about the MRS keyword <k>.

mrsload (mrsload <f>)
loads a file <f> of propositions.

mrssave (mrssave $<t_1> \ldots <t_n>$ <f>)
saves the propositions from theories $<t_1> \ldots <t_n>$ in the file <f> in a form that allows them to be reloaded with the mrsload command. Note that this works only for propositions stored in the pr representation.

mrstofunctions refers to the set of all MRS to<g> functions – e.g. (mem tostash mrstofunctions). See domain.

not (not <p>)
means that the proposition <p> is false. This is not equivalent to unknown, or to unprovable.

num-= (num-= <x> <y>)
means that the expressions <x> and <y> are numerically equal – or close enough to qualify. Procedurally, if <x> and <y> (are nonNIL and) unify, that MGU value is returned. Otherwise, if both terms are ground atomic numeric expressions, whose difference is less than NUM-=-THRESHOLD, truth is returned. E.g. (num-= 3 3.0) returns truth, whereas (unifyp 3 3.0) returns nil. See fea, num-=-theshold.

num-=-threshold is a special variable whose value is the tolerance required for two numeric values to be considered equal. It is initially set to 0.0001. See num-=.

number (number <x>)
means that <x> is a number.

or (or $<p_1> \ldots <p_n>$)
means that one or more of the propositions $<p_1> \ldots <p_n>$ is true.

output (output <x>)
translates the expression <x> into pseudo-natural language in accordance with programmer-defined templates. See template.

pattern (pattern <d>)
returns the proposition corresponding to the proposition symbol <d>.

perceive (perceive <p>)
determines whether the proposition <p> is true by direct observation rather than inference. Perceive is an abstract operator implemented using kb and toperceive.

perceive-indb (perceive-indb (indb <p>))

perceive-not (perceive-not (not <p>))

perceives (perceives <p>)
determines whether the proposition <p> is true by direct observation rather than inference and returns a list of all binding lists for which it succeeds. Perceives is an abstract operator implemented using kb and toperceives.

pl (repn <p> pl)
means that the proposition <p> should be represented in the pl representation, i.e. the pl-<x> family of subroutines will be used to stash, unstash, and lookup <p>. This representation is particularly useful for representing propositions involving unary functions, e.g. (arity member 2). Note that propositions stored in this way are not associated with any particular theory and cannot be found by PR-based routines like prfacts or prcontents. See pl-lookup, pl-stash, pl-unstash, repn.

pl-lookup (pl-lookup (<f> <a> <b>))
matches <b> against the <f> property of the lisp atom <a>. <f> and <a> must both be atoms. See pl.

pl-stash (pl-stash (<f> <a> <b>))
places <b> on the property list of <a> under the indicator <f>. <f> and <a> must both be atoms. See pl.

pl-unstash (pl-unstash (<f> <a> <b>))
removes the <f> property from the lisp atom <a>, if its value was <b>. <f> and <a> must both be atoms. See pl.

plug (plug <x> <l>)
returns a copy of the expression <x> fully instantiated with respect to the variables on the binding list <l>. For example, (plug (r $x $z) (($x . (f $y)) ($y . a))) would return (r (f a) $z).

pluralize (pluralize <x>)
returns the plural-value of <x>. That is, it returns (list <x>) if <x> is nonNIL, or nil otherwise. See lookupsbylookup, truepsbytruep.

pr (repn <p> pr)
means that the proposition <p> should be represented in the pr representation, i.e. the pr-<x> family of subroutines will be used to stash, unstash, and lookup <p>. This is MRSs default representation. Propositions stored in this way can be associated with any number of theories and will be available for use only when one or more of those theories are active. See pr-lookup, pr-indbp, pr-stash, pr-unstash.

pr-indbp (pr-indbp <p>)
checks whether there is a proposition in the pr representation that is identical to <p> up to variable renaming and, if so, returns its proposition symbol. See pr.

pr-lookup (pr-lookup <p>)
uses indexp and batchp to find any matching proposition in the pr representation and, if successful, returns the corresponding binding list. See pr.

pr-lookups (pr-lookups <p>)
uses indexp and batchp to find any matching proposition in the pr representation and returns a list of all binding lists for which it is successful. See pr.

pr-stash (pr-stash <p>)
stores <p> in the propositional data base and returns the corresponding proposition symbol. See pr.

pr-unstash (pr-unstash <p>)
removes the proposition <p> from the propositional data base. See pr.

prcontents (prcontents <th>)
prints out all propositions in theory <th>. Only those facts stored using the pr representation will be found.

preferred (preferred <j> <k>)
states that the task <j> is preferred to the task <k>. The preferred is

important in that <k> is disqualified whenever there is an applicable task that is preferred to it. This relation is the primary way of influencing task ordering in MRS. It has effect only when one of the switches executable or preferred has a non null value. See disqualified, scheduler.

prfacts (prfacts <n>)
prints out all propositions about <n> in the currently active theories. Only those facts stored using the pr representation will be found.

primitive (primitive <k>)
states that the operator in the task <k> is a primitive machine operation, i.e. a Lisp subroutine.

property (property <x> <y> <z>)
means that the atom <x> has <y> as its <z> property.

propositions (domain <x> <i> propositions)
means that the <i>th argument to the subroutine <x> should be a proposition. See domain.

provable (provable <p>)
means that proposition (provable <p>) is true if <p> can be proved using the normal mechanisms for proving <p>. See unprovable.

re (repn <p> re)
means that the proposition <p> should be represented in the re representation. That is, the re-<x> family of subroutines will be used to retrieve <p>. Its lookup method, re-lookup, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$) and calls lookupval on each of the $<x_i>$. If there is a LISP subroutine (i.e. on <r>s LISP property,) corresponding to the relation symbol <r>, re-lookup applies the subroutine and returns ((t p t)) if that subroutines returns nonNIL; otherwise it return nil. See computable-repn, repn, re-lookup, re-lookups, rq, reb, relnproc.

re-lookup (re-lookup <p>)
is used in trying to retrieve the proposition <p>. See re.

re-lookups (re-lookups <p>)
is functionally equivalent to (pluralize (Re-lookup p)). See re, re-lookup.

reb (repn <p> reb)
means that the proposition <p> should be represented in the reb representation. That is, the reb-<x> family of subroutines will be used to retrieve <p>. Its lookup method, reb-lookup, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$). If there is a LISP subroutine (i.e. on <r>s LISP property,) corresponding to the relation symbol <r>, reb-lookup applies the subroutine to these arguments, and returns the result (assumed to be a binding list). E.g. (repn (unifyp $a $b) reb). See computable-repn, repn, reb-lookup, reb-lookups.

reb-lookup (reb-lookup <p>)
is used in trying to retrieve the proposition <p>. See reb, relnproc.

reb-lookups (reb-lookups <p>)
is functionally equivalent to (pluralize (reb-lookup p)). See reb, reb-lookup.

rebm (repn <p> rebm)
means that the proposition <p> should be represented in the rebm representation. That is, the rebm-<x> family of subroutines will be used to retrieve <p>. Its lookup method, reb-lookup, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$). If there is a LISP subroutine (i.e. on <r>s LISP property,) corresponding to the relation symbol <r>, reb-lookup applies the subroutine to these arguments, and returns the result (assumed to be a list of binding lists). See computable-repn, reb, rebm-lookup, reb-lookups, relnproc, repn.

rebm-lookup (rebm-lookup <p>)
is functionally equivalent to (singularize (reb-lookup p)). See rebm, reb-lookup.

relnproc (2) (relnproc <sym> <op>)
is a way of procedurally attaching the operation <op> to the relation symbol <sym>. E.g. after asserting (relnproc < greater-than), (lookup (< 2 4)) will pass 2 and 4 to the LISP procedure greater-than, and seeing its answer is nonNIL, the lookup call will return ((t p t)). (It would otherwise return NIL.) The (relnproc <sym> <op>) assertion will use relnproc-assert to forward chain to assert that all propositions whose relation symbol is <sym> should use the rq representation. See funproc, relnproc (3), relnproc-assert, rq.

relnproc (3) (relnproc <sym> <op> <repn>)
There are various possible limitations with binary relnproc mechanism, listed above under relnproc (2). First, after asserting (relnproc < greater-than), the query (lookup (< 2 (+ 1 3))) will return nil. Second, after asserting (relnproc unify unifyp), (lookup (unify (a $y) ($x b))) will only return ((t p t)), ignoring the ($x . a) and ($y . b) bindings. The (optional) third argument above, <repn>, permits other, more elaborate forms of relational procedural attachment. The (relnproc < greater-than RE) assertion handles the first problem. Here each embedded ground term – here 2 and (+ 1 3) – will first be lookupvaled, and the result passed to the LISP procedure greater-than. This uses the re representation, rather than rq. The (relnproc unify unifyp RQB) assertion handles the second problem, as the (unify &x &y) facts will now use the RQB representation. Similarly <repn> can be set to RQBM, REB, REQB or RQFM. If omitted, <repn> here defaults to rq. One can also use the aliases EVAL for RE, BindList for RQB, MBindList for REBM, EBindList for REB, MEBindList for REBM, or MultiFn for RQFM. See relnproc (2), relnproc-assert, re, reb, rebm, rq, rqb, rqbm, rqfm.

relnproc-assert (relnproc-assert (relnproc <sym> <op> <repn>))
asserts the proposition (relnproc <sym> <op> <repn>). (The same subroutine is used to assert (relnproc <sym> <op>).) See relnproc.

repn (repn <p> <rpn>)
means that the representation <rpn> should be used to store and access the proposition <p>. (See repns entry for list of allowable representations.) See achieve, repn-assert, repn-method, repn-unassert, repns.

repn-assert (repn-assert (repn <prop> <rpn>))
uses the repn-method declarations associated with this representation <rpn> to stash (in a forward chaining manner) the appropriate to<x> statements for this <prop>. The domain specification of that operation is used to determine the exact form of the assertion. E.g. calling repn-assert on the assertion (repn (father &c &d) pl) will generate the statements (tolookup (father &c &d) pl-lookup), and (tolookups (father &c &d) pl-lookups), as (domain lookup 1 propositions). (Later it may deal with terms, and assert facts like (tolookupval (father &c) pl-lookupval), (via (domain lookupval 1 terms)) as well.) See domain, repn, repn-method, repn-unassert.

repn-method (repn-method <rpn> <op> <mthd>)
means that the LISP subroutine <mthd> should be used to perform the <op> operation, in the representation <rpn>. E.g. (repn-method pr tostash pr-stash). See repn-assert, repn, repn-unassert.

repn-unassert (repn-unassert (repn <prop> <rpn>))
undoes the effects (read stashes) of repn-assert. See domain, repn, repn-assert, repn-method.

repns (mem <r> repns)
means that the symbol r refers to a representations. Currently existing representations include cnf, dl, pl, pr, tl and achieve-perceive, all of which store results; and fe, fes, fq, re, reb, rebm, rq, rqb, rqbm and rqfm which do not. See repn, computable-repn.

residue (residue <p>)
tries to prove the proposition <p>. It differs from truep in that it is allowed to make assume any proposition asserted to be assumable; and, if it is successful in proving <p>, it returns a list of its assumptions. The set of assumptions is called the residue of <p>. Residue is an abstract operator defined using kb and toresidue. See assumable.

residues (residues <p>)
tries to prove the proposition <p>. It differs from trueps in that it is allowed to make assume any proposition asserted to be assumable; and, if it is successful in proving <p>, it returns a list of lists of assumptions. The set of assumptions is called the residue of <p>. Residue is an abstract operator defined using kb and toresidue. See assumable.

resolution **(resolution <p>)**
tries to prove the proposition <p>. If successful, it returns an appropriate binding list; otherwise, it returns **nil**. **Resolution** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolution** negates <p> and converts it to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rsdisp** on each conjunct. See **rsdisp**, **scheduler**, **tracetask**, **cnf**.

resolutionresidue **(resolutionresidue <p>)**
tries to prove the proposition <p>. If succesful, it returns a list of assumable propositons which, when added to the data base, imply <p>. **Resolutionresidue** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutionresidue** negates <p> and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rrdisp** on each conjunct. See **rrdisp**, **scheduler**, **tracetask**, **cnf**.

resolutionresidue **(resolutionresidue <p>)**
tries to prove the proposition <p>. If succesful, it returns a list of assumable propositons which, when added to the data base, imply <p>. **Resolutionresidue** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutionresidue** negates <p> and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rrdisp** on each conjunct. See **rrdisp**, **scheduler**, **tracetask**, **cnf**

resolutionresidues **(resolutionresidues <p>)**
tries to prove the proposition <p>. It returns a list of all assumption lists for which it is successful. **Resolutionresidues** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutionresidues** negates <p> and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rrdisp** on each conjunct. See **rrdisp**, **scheduler**, **tracetask**, **cnf**.

resolutions **(resolutions <p>)**
tries to prove the proposition <p>. It returns a list of all binding lists for which it is successful. **Resolutions** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutions** negates <p> and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rsdisp** on each conjunct. See **rsdisp**, **scheduler**, **tracetask**, **cnf**.

rq **(repn <p> rq)**
means that the proposition <p> should be represented in the **req** representation. That is, the **rq-<x>** family of subroutines will be used to retrieve <p>. Its lookup method, **rq-lookup**, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$). If there is a LISP subroutine corresponding to the relation symbol <r>, (i.e. on

<r>s LISP property,) rq-lookup applies the subroutine to the arguments (<x_1 > ... <x_n >), and returns truth if the result is nonnil; or nil. By default, most relations, including arithmetic ones, use this representation. See rs, rq-lookup, rq-lookups, relnproc, repn.

rq-lookup — (rq-lookup <p>)
is used in trying to retrieve the proposition <p>. See rq, lisp.

rq-lookups — (rq-lookups <p>)
is functionally equivalent to (pluralize (Rq-lookup p)). See rq, rq-lookup.

rqb — (repn <p> rqb)
means that the proposition <p> should be represented in the rqb representation. That is, the rqb-<x> family of subroutines will be used to retrieve <p>. Its lookup method, rqb-lookup, takes as its argument a proposition of the form (<r> <x_1 > ... <x_n >). If there is a LISP subroutine corresponding to the relation symbol <r>, (i.e. on <r>s LISP property,) rqb-lookup applies the subroutine to the arguments (<x_1 > ... <x_n >), and simply returns the result (assumed here to be a binding-list). See computable-repn, rsb, rqb-lookup, rqb-lookups, relnproc, repn.

rqb-lookup — (rqb-lookup <p>)
is used in trying to retrieve the proposition <p>. See rqb, rqbm, lisp.

rqb-lookups — (rqb-lookups <p>)
is functionally equivalent to (pluralize (Rqb-lookup p)). See rqb, rqb-lookup.

rqbm — (repn <p> rqbm)
means that the proposition <p> should be represented in the rqbm representation. That is, the rqbm-<x> family of subroutines will be used to retrieve <p>. Its lookup method, rqb-lookup, takes as its argument a proposition of the form (<r> <x_1 > ... <x_n >). If there is a LISP subroutine corresponding to the relation symbol <r>, (i.e. on <r>s LISP property,) rqb-lookup applies the subroutine to the arguments (<x_1 > ... <x_n >), and simply returns the result (assumed here to be a list of binding-lists). See computable-repn, rqb, rqbm-lookup, rqb-lookup, relnproc, repn.

rqbm-lookup — (rqbm-lookup <p>)
is functionally equivalent to (singularize (Rqb-lookup p)). See rqbm, rqb-lookup.

rqfm — (repn <p> rqfm)
means that the proposition <p> should be represented in the rqfm representation. That is, the rqfm-<x> family of subroutines will be used to retrieve <p>. Its lookup method, rqfm-lookups, takes as its argument a proposition of the form (<r> <x_1 > ... <x_n > <x_{n+1} >). It first evaluates the term (<r> <x_1 > ... <x_n >) by applying LISP

subroutine corresponding to the function symbol <r>, (i.e., on <r>s LISP property,) to the list $<x_1> \ldots <x_n>$. This returns a list of values. Rqfm-lookups then unifies each of these with $<x_{n+1}>$, returning the list of results. See computable-repn, rqfm-lookup, rqfm-lookups, relnproc, repn.

rqfm-lookup (rqfm-lookup <p>)
is functionally equivalent to (singularize (Rqfm-lookups p)). See rqfm, rqfm-lookups.

rqfm-lookups (rqfm-lookups <p>)
is used to retrieve the propositions <p>. See rqfm, lisp.

rrdisp (rrdisp <l> <cl> <al> <th>)
performs one resolution step in trying to derive a contradiction from the propositions on the list <l>. The list <cl> contains a list of assumptions made in preceding steps. The binding list <al> holds the bindings of the variables from preceding steps. The theory <th> holds the conjuncts from the negated goal. To be used, all propositions must be in conjunctive form. In addition, only base-level variables are treated as variables, any meta-level variables are treated as constants. Given a goal list (<p> . <l>), rrdisp generates subgoals by negating <p> to get <q> and looking in the data base for propositions of the form <q> or (or ... <q> ...). It also uses trueps to discover whether p is assumable. If it is assumable and if it is a ground proposition after plugging in the variable bindings returned by trueps, brdisp creates a new theory that includes <th>, asserts the proposition in that theory, and generates appropriate subgoals. The asserted propositions are useful in that they make possible consistency checks before making assumptions in subsequent steps. The order in which multiple rrdisp tasks are executed can be influenced via appropriate preferred propositions. See assumable, cnf.

rsdisp (rsdisp <l> <al> <th>)
performs one resolution step in trying to derive a contradiction from the propositions on the list <l>. The binding list <al> holds the bindings of the variables from preceding steps. The theory <th> holds the conjuncts from the negated goal. To be used, all propositions must be in conjunctive form. In addition, only base-level variables are treated as variables, any meta-level variable is treated as a constant. Given a goal list (<p> . <l>), rsdisp generates subgoals by negating <p> to get <q> and looking in the data base for propositions of the form <q> or (or ... <q> ...). The order in which multiple rsdisp tasks are executed can be influenced via appropriate preferred propositions. See cnf.

runnable (runnable <k>)
states that the task <k> is runnable. A runnable task is applicable if its operator is a Lisp subroutine; otherwise, it is assumed to be a defined task, and a corresponding invocation of exec is applicable. In

MRS demons are implemented via runnable, e.g. to tell the system that it should print a greeting whenever the user asserts a proposition that someone is logged, one simply asserts (if (loggedin $x) (runnable (print hello t))) and (toassert (loggedin &x) fc). See applicable, scheduler.

samep (samep <x> <y>)
determines whether expressions <x> and <y> are the same under consistent variable renaming, and if so returns a binding list for the variables in <x>. For example, (p $x $y $x) is the same as (p $y $x $y) but not (p $x $y $y).

scheduler (scheduler)
Scheduler is the heart of the MRS system. It is a simple deliberation-action loop that, at each point in time, decides on an executable task, executes it, and repeats. In its most general state, the choice of task is made by calling trtruep to find a task <k> such that (executable <k>) is true. After the task is performed, the fact is recorded by calling trassert on the proposition (executed <k>). In its initial state, MRS contains a number of propositions to help scheduler decide on an executable task. In particular, a task is executable if it is applicable and is not disqualified. Once a task becomes applicable, it remains applicable until it is executed. One way an applicable task can be disqualified is for there to be another applicable task that is preferred to it. A runnable task is applicable if its operator is a Lisp subroutine. Otherwise, it is assumed to be a defined task, and a corresponding exec task is applicable.
This full generality is available only if the switches executable and executed are both non-nil. For reasons of efficiency, both of these switches are initially set to nil, and an optimised version of this loop is used instead. In particular, the set of applicable tasks is kept as the value of the variable agenda, and an executable task is obtained from this list. If the switch preferred is nil, the first element of the list is taken; otherwise, scheduler uses trtruep to compare the elements using the preferred relation. This optimisation is fully consistent with the axioms described above. However, it is recommended that the user not change the settings of executable and executed without careful forethought. Debugging facilities for the scheduler architecture are not very good at this point. However, rudimentary debugging is possible using tracetask.

setdiff (setdiff <x> <y> <b>)
means that list <b> is all the elements in list <x> that are not in list <y>.

```
(setdiff nil $y nil)
(if (and (not (element $e $y)) (setdiff $l $y $s))
    (setdiff ($e . $l) $y ($e . $s)))
(if (and (element $e $y) (setdiff $l $y $s))
```

```
(setdiff ($e . $l) $y $s))
```

Procedural attachment: truep-setdiff. The lisp file set must be loaded from the mrs directory.

setof (setof <x> <p> <s>)
means that <s> is the set of all objects <x> that satisfy <p>.

```
(if (and (bagof $x $p $b) (elementsin $b $s))
    (setof $x $p $s))
```

Procedural attachments: truep-setof and lookup-setof. The lisp file set must be loaded from the mrs directory.

singlep (singlep <p>)
returns t when the proposition <p> has at most one solution, i.e. when it is a ground proposition or an atomic proposition with a functional operator and ground arguments. See function.

singularize (singularize <x>)
returns the singular-value of <x>. That is, it returns (car <x>). See lookupbylookups, truepbytrueps.

stash (stash <p>)
stores the proposition <p> in the data base. Stash is an abstract operator implemented using kb and tostash.

stash-and (stash-and (and $<p_1> \ldots <p_n>$))
separately stashes each of the conjuncts $<p_1>, \ldots, <p_n>$.

stashapplicable (stashapplicable (applicable <k>))
adds <k> to agenda. See applicable.

subclass (subclass $<c_1>$ $<c_2>$)
means that the set $<c_2>$ is a subset of $<c_1>$ – that is, all members of $<c_2>$ are members of $<c_1>$. E.g., (subclass number integer). See classes.

subset (subset <x> <y>)
means that every element of the list <x> is an element of the list <y>.

```
(subset nil $y)
(if (and (element $e $y) (subset $l $y))
    (subset ($e . $l) $y))
```

Procedural attachment: truep-subset. The lisp file set must be loaded from the mrs directory.

succeed (succeed <z>)
is a special control form. Executing this form causes the enclosing doable task to succeed or the enclosing undoable task to fail. In addition, all other subtasks are discarded.

task In MRS the task of performing an operator <op> with arguments $<x_1>, \ldots, <x_n>$ is written (<op> $<x_1> \ldots <x_n>$. The operator in task <k> may be a Lisp subroutine, an MRS subroutine defined using def, or a special control form like doand, doable, or doall. If

the operator is a Lisp subroutine, the task must have have an additional argument for the output value, and it will succeed only if the last argument unifies with the result of calling the subroutine on all but the last argument. For example, the task (cdr (a b c) (&x . &y)) will succeed with the variable &x bound to b and the variable &y bound to (c). See execute, executes.

tb (tb <op> <x1> ... <xn>)
makes the task (<op> <x1> ... <xn>) applicable by placing it on the agenda. See applicable, agenda.

template (template <x> <t>)
means that expression <x> should be output as <t>. In particular, if an expression <y> matches <x> with binding list <al>, (output <x>) returns a copy of <t> in which each variable is replaced by the result of calling output on its value in <al>. Templates are used by output.

terms (domain <x> <i> terms)
means that the <i>th argument to the subroutine <x> should be a term. See domain.

test (test <file>)
runs the single test file provided <file> and prints out any errors. Returns the number of errors found in the file.

thassert (thassert <p> <th>)
binds theory to <th> and asserts the proposition <p>. See assert, theory.

theories (domain <x> <i> theories)
means that the <i>th argument to the subroutine <x> should be a theory. See domain.

theory has as its value the name of the current theory. All propositions stored in the data base via pr-stash are associated with the theory named as the value of theory at the time of the stash. One can associate a proposition with more than one theory by repeating the call to pr-stash with different values for theory. Calling pr-unstash removes a proposition only from the current theory. The theory named as the value of theory is always active, i.e. the propositions associated with it are available for retrieval by pr-lookup and pr-lookups. See pr-stash, pr-unstash, pr-lookup, and pr-lookups.

thfalse (thfalse (unprovable <p>))
calls truep on the proposition <p>. It returns nil if the answer is non-nil; otherwise, it returns truth.

threpn (threpn <p> <rpn> <th>)
means that the representation <rpn> should be used to store and access the proposition <p> when <th> is an active theory. The effect of having conflicting representations for a proposition stored in different

theories is undefined when both theories are active. (See repns entry for list of allowable representations.) See activate, deactivate, theory, achieve, repns, repn, repn-assert, repn-method, repn-unassert.

thstash (thstash <p> <th>)
binds theory to <th> and stashes the proposition <p>. See stash, theory.

thtrue (thtrue (provable <p>))
calls truep on the proposition <p> and returns the answer.

thunassert (thunassert <p> <th>)
binds theory to <th> and unasserts the proposition <p>. See unassert, theory.

thunstash (thunstash <p> <th>)
binds theory to <th> and unstashes the proposition <p>. See unstash, theory.

tl (repn <p> tl)
means that the proposition <p> should be represented in the tl representation, i.e. the tl-<x> family of subroutines will be used to stash, unstash, and lookup <p>. This representation is particularly useful for storing propositions involving unary relations, e.g. (function f). Note that propositions stored in this way are not associated with any particular theory and cannot be found by PR-based routines like prfacts or prcontents. See repn, tl-lookup, tl-stash,tl-unstash

tl-lookup (tl-lookup (<r> <a>)
returns truth if there is an <r> property on the atom <a>. Both <r> and <a> must be atoms. See tl.

tl-stash (tl-stash (<r> <a>))
sets the <r> property of the lisp atom <a> to t. Both <r> and <a> must be atoms. See tl.

tl-unstash (tl-unstash (<r> <a>))
removes the <r> property of <a>. Both <r> and <a> must be atoms. See tl.

tm-unassert (tm-unassert <p>)
calls unstash on <p> and then calls unassert on any proposition all of whose justifications depend on <p>. See just.

to<g> (to<g> <p> <f>)
means that the subroutine <f> is to be called in performing the action <g> on argument <p>. Each of MRSs user-level commands has associated with it a relation that specifies the subroutine to be used in carrying out that command. The relation is named by prefixing the commands name with to, e.g. toAssert from assert. See kb.

toachieve (toachieve <p> <m>)
means that the method <m> should be used to perform the achieve ac-

tion for all propositions which match <p>. See kb, achieve, to<x>.

toassert (toassert <p> <m>)
means that the method <m> should be used to perform the assert action for all propositions which match <p>. See kb, assert, to<x>.

tocache (tocache <p> <m>)
means that the method <m> should be used to cache propositions which match <p>. (This <m> will only be used when the variable cache has a nonNIL value.) See cache, cachebystash, to<x>

tolookup (tolookup <p> <m>)
means that the method <m> should be used to determined the lookup value for all propositions which match <p>. See kb, lookup, to<x>, tolookups.

tolookups (tolookups <p> <m>)
means that the method <m> should be used to determined the lookups values for all propositions which match <p>. See kb, lookups, to<x>, tolookup.

toperceive (toperceive <p> <m>)
means that the method <m> should be used to determined the perceive value for all propositions which match <p>. See kb, perceive, to<x>, toperceives.

toperceives (toperceives <p> <m>)
means that the method <m> should be used to determined the perceives values for all propositions which match <p>. See kb, perceives, to<x>, toperceive.

toplevel (toplevel)
is a read-execute-print loop. See execute.

tostash (tostash <p> <m>)
means that the method <m> should be used to perform the stash action for all propositions which match <p>. See kb, stash, to<x>.

totruep (totruep <p> <m>)
means that the method <m> should be used to determined the truep value for all propositions which match <p>. See kb, truep, to<x>, totrueps.

totrueps (totrueps <p> <m>)
means that the method <m> should be used to determined the trueps values for all propositions which match <p>. See kb, trueps, to<x>, totruep.

tounachieve (tounachieve <p> <m>)
means that the method <m> should be used to perform the unachieve action for all propositions which match <p>. See kb, unachieve, to<x>.

tounassert **(tounassert <p> <m>)**
means that the method <m> should be used to perform the unassert action for all propositions which match <p>. See kb, unassert, to<x>.

tounstash **(tounstash <p> <m>)**
means that the method <m> should be used to perform the unstash action for all propositions which match <p>. See kb, unstash, to<x>.

tracetask **(tracetask <p>)**
As each task is executed, tasktrace prints out the name of the subroutine and its arguments provided they match <p>. If there is no <p> argument in the subroutine call then a list of all tasks which are to be traced is printed. See untracetask.

trassert **(trassert <p>)**
asserts the proposition <p> and performs forward chaining as appropriate. Trassert is MRSs meta-level assertion routine and is called by many MRS subroutines. The default is simply to stash a proposition, but there are also built-in procedural attachments for propositions containing certain special relations (stored on each relation as the assert property). A frequently used procedural attachment is the depth-first forward chaining program trfc.

trlookup **(trlookup <p>)**
looks up the proposition <p>. If successful, it returns the corresponding binding list; otherwise, it returns nil. Trloookup is one of MRSs meta-level lookup routines and is called by many MRS subroutines. The default procedure uses indexp and matchp to find any matching propositions in the pr representation, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its lookup or lookups property).

trlookups **(trlookups <p>)**
looks up the proposition <p> and returns a binding list for each matching proposition that it finds. Trlookups is one of MRSs meta-level lookup routines and is called by many MRS subroutines. The default procedure uses indexp and matchp to find any matching propositions in the pr representation, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its lookup lookups property).

trstash **(trstash <p>)**
stashes the proposition <p>. Trstash is MRSs meta-level stash routine and is called by many MRS subroutines. The default is pr-stash, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its stash property).

trtruep **(trtruep <p>)**
tries to prove the proposition <p>. If successful, it returns a corresponding binding list; otherwise, it returns nil. Trtruep is one of MRSs meta-level theorem proving routines and is called by many MRS subrou-

tines. Only meta-level variables are treated as variables by trtruep, and all base-level variables are treated as constants. The inference procedure used is the depth-first backward chaining program trbc, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its truep or trueps property).

trtrueps (trtrueps <p>)
tries to prove the proposition <p> and returns a list of all binding lists for which it is successful. Trtrueps is one of MRSs meta-level theorem proving routines and is called by many MRS subroutines. Only meta-level variables are treated as variables by trtrueps, and all base-level variables are treated as constants. The inference procedure used is the depth-first backward chaining program trbcs, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its truep or trueps property).

truep (truep <p>)
tries to prove the proposition <p>. If it is successful, it returns a binding list for the base-level variables in <p>; otherwise, it returns nil. Truep is an abstract operator implemented using kb and totruep.

truep-bagof (truep-bagof (bagof <x> <p> <s>))
calls trueps on <p> and matches <s> against the list formed by plugging the answers into <x>.

truep-is (truep-is (is <x> <y>))
uses getval to evaluate the arbitrarily nested expression <x> and tries to unify the answer with <y>. See is.

truepbytrueps (truepbytrueps <p>)
is equivalent to (singularize (trueps <p>)).

trueps (trueps <p>)
tries to prove the proposition <p> and returns a list of all binding lists for which it is successful. Trueps is an abstract operator implemented using kb and totrueps.

truepsbytruep (truepsbytruep <p>)
is equivalent to (pluralize (truep <p>)).

trunassert (trunassert <p>)
unasserts the proposition <p>. Trunassert is MRSs meta-level unassertion routine and is called by many MRS subroutines. The default is simply to unstash a proposition, but there are also built-in procedural attachments for propositions containing certain special relations (stored on each relation as the unassert property).

trunstash (trunstash <p>)
unstashes the proposition <p>. Trunstash is MRSs meta-level unstash routine and is called by many MRS subroutines. The default is prunstash, but there are also built-in procedural attachments for proposi-

tions containing many common relations (stored on each relation as its unstash property).

truth has ((t p t)) as its value. The value of truth occurs as the last pair in binding lists returned by MRSs retrieval and inference procedures.

tutor (tutor)
runs an interactive tutor that introduces one to the basic representation and inference mechanisms of MRS.

unassert (unassert <p>)
removes the proposition <p> from the data base and performs all appropriate inference. Unassert is an abstract operator implemented using kb and tounassert.

unassert-and (unassert-and (and $<p_1>$ ... $<p_n>$))
separately unasserts each of the conjuncts $<p_1>$,..., $<p_n>$.

unassert-iff (unassert-iff (if <p> <q>))
asserts (if <p> <q>) and (if <q> <p>).

undoable (undoable <k>)
designates the task of trying to execute the task <k>. The task (undoable <k>) succeeds only if there is no successful execution of <k>. Note that as a result of the current implementation, it is not possible to interleave subtasks outside a doable task with those inside. See succeed and cut.

unifyp (unifyp <x> <y>)
determines whether expressions <x> and <y> are unifiable, and if so returns their most general unifier. Unifyp differs from matchp in that multiple occurrences of the same variable in both <x> and <y> are not treated as distinct variables. For example, (p $x b) and (p a $x) are not unifiable, but they do match.

unincludes (unincludes $<t_1>$ $<t_2>$)
removes any includes link between theories $<t_1>$ and $<t_2>$. See includes.

union (union <x> <y> <b>)
means that list <b> is the lists <x> and <y> appended together.

```
(union nil $y $y)
(if (union $l $y $s)
    (union ($x . $l) $y ($x . $s)))
```

Procedural attachment: truep-union. The lisp file set must be loaded from the mrs directory.

unknown (unknown <p>)
means that if proposition <p> is not in the database then (unknown <p>) is true. See known.

unprovable (unprovable <p>)
means that proposition (unprovable <p>) is true if <p> cannot be

proved using the normal mechanisms for proving <p>. See provable.

unstash (unstash <p>)
removes the proposition <p> from the data base. Note this is not equivalent to asserting the negation of <p>. Unstash is an abstract operator implemented using kb and towerstash.

unstash-and (unstash-and (and $<p_1>$ ... $<p_n>$))
separately unstashes each of the conjuncts $<p_1>,\ldots,<p_n>$.

unstashapplicable (unstashapplicable (applicable <k>))
removes <k> from agenda. See applicable.

untracetask (untracetask <p>)
untraces the task <p>. If there is no <p> argument in the subroutine call then it untraces all <p> that are currently being traced. See tracetask.

value (value <x> <y>)
means that the atom <x> has value <y>.

variable (variable <x>)
means that the symbol <x> is a variable.

varp (varp <xp>)
returns a non-nil value if <xp> is a variable and otherwise returns nil. See blvarp and mlvarp.

where (where <p>)
prints out a message for each recorded justification in which <p> is a premise. The message includes information about the justified proposition, the inference method, and all premises. See justify, just.

why (why <p>)
prints out a message for each recorded justification for the proposition <p>. The message includes information about the relevant inference method and all premises. See justify, just.

INDEX

Page numbers given in bold type indicate a definition entry.

END
FILMED

DATE:
4-17-96

NTIS